P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks

Apostolos Kokolis, Thomas Shull, Jian Huang and Josep Torrellas
University of Illinois at Urbana-Champaign
{kokolis2, shull1, jianh, torrella}@illinois.edu

***Abstract*—The availability of user-friendly programming frameworks is key to the success of Non-Volatile Memory (NVM). Unfortunately, most current NVM frameworks rely heavily on user intervention to mark persistent objects and even persistent writes. This not only complicates NVM programming, but also introduces potential bugs. To address these issues, researchers have proposed Persistence by Reachability frameworks, which require minimal user intervention. However, these frameworks are slow because their runtimes have to perform checks at program load/store operations, and move data structures between DRAM and NVM during program execution.**

In this paper, we introduce P-INSPECT, a novel hardware architecture targeted to speeding up persistence by reachability NVM programming frameworks. P-INSPECT uses bloom-filter hardware to perform various checks in a transparent and efficient manner. It also provides hardware for low-overhead persistent writes. Our simulation-based evaluation running a state-of-the-art persistence by reachability framework shows that P-INSPECT retains programmability and eliminates most of the overhead. We use real-world applications to demonstrate that, on average, P-INSPECT reduces an application's number of executed instructions by 26% and the execution time by 16%—delivering similar performance to an ideal runtime that has no persistence by reachability overhead.

***Index Terms*—Non-volatile memory, Programming frameworks, Hardware for programmability**

I. INTRODUCTION

Byte-addressable Non-Volatile Memory (NVM) technologies such as 3D XPoint [1], Phase Change Memory (PCM) [2, 3, 4], and Resistive RAM (ReRAM) [5] have recently gained much attention. They offer high storage density, low static power, non-volatility, and performance characteristics that are comparable to those of DRAM [6]. Thanks to these properties, NVM is expected to create disruptive changes to many application domains and software systems.

To a large extent, the success of NVM depends on the availability of user-friendly programming frameworks for software development [7]. For this reason, many NVM programming frameworks have been proposed (e.g., [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]). Although they have different implementations, they share the same challenges, including what persistence abstractions to provide, how to identify the objects to allocate in NVM versus DRAM, and how to identify the stores that modify the persistent state.

Most of the frameworks rely heavily on programmer involvement. They require programmers to mark all the objects that must be allocated in NVM (e.g., [10, 15, 16, 9, 18, 11, 13, 14]). Many frameworks also require programmers to identify the stores that modify NVM (e.g., [10, 11, 12, 13, 14, 15, 17, 20]), and potentially augment the code with instructions that write back a cache line to NVM (*CLWB*) [26], order instructions (store fence or *sfence*), and log state in NVM. This results in programming difficulty, introduces software bugs, and generates nonreusable code.

Ideally, NVM frameworks should assume all of these aforementioned responsibilities. One class of NVM frameworks that come close to this ideal is *Persistence by Reachability* frameworks (e.g., [21, 22, 23]). The idea is that the programmer only identifies the *durable roots*—i.e., the few entry points into the program's data structures that should reside in NVM. There is no need to mark all the persistent objects. Then, during execution, the runtime software ensures that all the data structures that are reachable from the durable roots are crash-consistent. The runtime does so by moving the data structures to NVM when needed, identifying persistent stores and adding CLWB and sfence instructions, and performing logging when required.

Unfortunately, while these frameworks are user-friendly, their runtime adds substantial performance overhead [21, 27]. Since the properties of program structures change dynamically, the framework has to perform checks at every program load and store, and move data structures between DRAM and NVM at runtime. To make persistence by reachability frameworks attractive to the community, and the paradigm of choice among programmers, they must have lower overhead.

The operation of such frameworks dictates that they possess fine-grain dynamic information about the program's data structures, such as memory location and persistence properties. Currently, there is no efficient hardware technique to provide such information. Approaches like Intel's MPX [28] can only provide limited information (i.e., pointer bounds) about program structures, while approaches that rely on tagging memory locations [29, 30, 31] incur too much overhead to be used in production code.

In this paper, we introduce novel hardware support targeted to accelerate NVM programming frameworks that provide persistence by reachability. Our scheme, named P-INSPECT, focuses on reducing the main source of overhead in these NVM frameworks: state checks performed before read and write accesses. Specifically, in P-INSPECT, an application read/write includes inexpensive hardware checks of the state of

This work was funded in part by NSF under grants CNS 1956007, CNS 1763658, and CCF 1725734.

the accessed data structure. In the common case, the hardware checks conclude that no special action is needed, and the read/write completes normally. Otherwise, a runtime software handler is automatically invoked, which performs any needed framework operations. To perform these checks efficiently, P-INSPECT uses cache-coherent hardware bloom filters.

In addition, P-INSPECT also speeds up the execution of persistent writes. It does so by combining writes with CLWB and sfence instructions in hardware.

Our evaluation shows that P-INSPECT retains the programmability advantages of persistence by reachability frameworks while removing most of their execution overhead. Specifically, we run a key-value store with various Yahoo Cloud Service Benchmarks, and several kernels on a state-of-the-art persistence by reachability framework. With P-INSPECT hardware support, real-world applications reduce their number of executed instructions and their execution time by an average of 26% and 16%, respectively. We also compare P-INSPECT to an ideal runtime that has no persistence by reachability overhead, and demonstrate similar performance improvement.

This work makes the following contributions:

- We propose P-INSPECT, the first hardware architecture to accelerate persistence by reachability NVM frameworks in a transparent manner.
- We develop hardware designs to minimize persistence checks in software and to speed up persistent writes.
- We evaluate the proposed hardware on a state-of-the-art persistence by reachability NVM framework, and demonstrate its improved performance.

II. Background: Using NVM

Programming an NVM system is challenging [32, 7]. To use NVM, programming frameworks must offer users an appropriate interface. For instance, the Storage Networking Industry Association (SNIA) has created a low-level programming model for NVM developers [24]. Intel has produced a tool set that is compatible with the SNIA model, which is called Persistent Memory Development Kit (PMDK) [25], and is a collection of libraries in C/C++ and Java. In addition to the SNIA and Intel efforts, there are many other NVM programming frameworks [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. Their goal is to facilitate NVM programming and ease the adoption of NVM.

The two main differentiating features of these frameworks are: (i) how they identify which data objects to place in NVM, and (ii) what persistence abstractions they provide. Many frameworks [10, 15, 16, 9, 18, 11, 13, 14, 17, 25] expect the user to explicitly mark all the objects that must be placed in NVM. Such a limited abstraction places a heavy burden on the programmer and creates many opportunities for bugs [32, 33, 34]. Furthermore, these frameworks cannot be directly used with existing codes; they require applications to be rewritten to include the necessary markings.

In contrast, several new frameworks [21, 22, 23] use a more programmer-friendly model known as *Persistence by Reachability*. In these frameworks, the runtime automatically ensures that, given a set of entry points into the persistent data structures in the program, all the data in the program reachable from such entry points is in NVM. This support significantly reduces the programmer burden and allows for the reuse of existing code.

Frameworks also differ in the persistence abstractions provided. Frameworks supporting epochs [12, 11, 13, 14, 20] couple logging regions to epochs (e.g., critical sections), and stores only need to be persisted at the end of the epochs. Other frameworks allow the logging regions to be specified by the programmer, and are not tied to a programming construct [9, 10, 15, 16, 17, 21, 22, 23]

Another difference between frameworks is whether the user is expected to explicitly identify the persistent stores in the program. Some frameworks expect the user to mark persistent stores manually [10, 11, 12, 13, 14, 15, 18, 9, 25], introducing many opportunities for mistakes. However, others leverage both compile time and runtime techniques [16, 8, 19, 21, 22, 23, 20, 17] to obviate the need for the programmer to label persistent stores. As persistence by reachability NVM frameworks already leverage runtime techniques, they free the programmer from having to label persistent stores.

III. Persistence by Reachability

A. Benefits of the Reachability Abstraction

In *Persistence by Reachability* [21, 22, 23], the programmer is only tasked with identifying the few entry points into the data structures that should be made persistent. These are called *durable roots*, and could be, e.g., the dominator pointer to a graph structure, or the root node of a tree. The framework is responsible to ensure that, dynamically, all objects reachable from a durable root (i.e., the durable root's *transitive closure*) reside in NVM.

This model is attractive because it puts a minimal burden on programmers. Programmers only have to think about which data structures must be in NVM, instead of explicitly identifying all the objects that must be in NVM. Throughout execution, data structures are modified and the durable transitive closure changes. Therefore, the framework monitors program writes and moves objects to NVM as needed.

These frameworks have three other advantages. First, they simplify programming by automatically inserting the necessary CLWBs, sfences, and logging operations. Second, they are compatible with existing code, in that neither the code must be rewritten to identify the persistent state nor a different library must be used. Finally, by dynamically placing data in NVM only when needed, they save power and latency.

B. Reacting to Updates to the Durable Transitive Closure

At runtime, these frameworks guarantee that the durable root set's transitive closure always resides in NVM. This requires objects to be moved to NVM as they become reachable from a durable root. To understand the operations needed, Figure 1(a) shows a heap containing objects A, B, and C in DRAM, and E and F in NVM. F is a durable root. An

arrow from object *i* to *j* means that a field in *i* has a reference to *j*. Each object has a header state with 2 bits: the *Forwarding* and *Queued* bits. Their functionality is explained below.

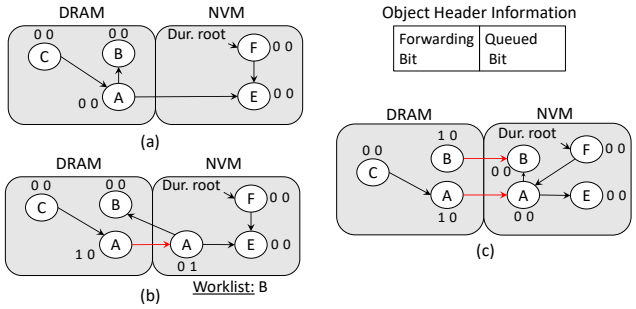

Fig. 1: Persistence by reachability example.

Suppose that F is updated to point to A instead of E. To ensure that the durable transitive closure is completely within NVM, before performing the write that makes F point to A, the framework must transparently move both A and its transitive closure (i.e., the objects pointed by A, recursively) to NVM. This is done iteratively, using a worklist of objects starting with A. For each object obj in the worklist (e.g., A), the framework does:

1) Create a copy of obj in NVM with a set Queued bit.
2) Update the original obj by setting its Forwarding bit and making it point to the object's new location.
3) Search obj's fields for references to other objects to be added to the worklist.

In the first step, the object is copied to NVM with a Queued bit set to indicate that the object's transitive closure is still being processed. All the objects in the transitive closure being moved will set their Queued bit, which will only get reset once there are no more objects within the worklist to move to NVM.

Setting the Queued bit is necessary for correct execution in multithreaded environments. It ensures that another thread does not prematurely update a field somewhere to point to an object copied to NVM before the object's transitive closure is made durable. Hence, any write must check the Queued bit and wait to perform until the bit is reset.

Step two repurposes the original object to act as a *Forwarding* object pointing to the object's new NVM location. Forwarding objects are essential to allow pointers in DRAM to be lazily updated. The alternative would be to stop the execution of all the threads and update all the pointers that point to the old object in DRAM to point to the new object in NVM. This is eschewed by persistence by reachability frameworks because it would have prohibitive performance overheads. Instead, when accessing an object, the framework first checks whether the object is a forwarding object by checking its Forwarding bit. If it is, the NVM location of the object is accessed. Note that forwarding objects are always in DRAM and point to NVM.

The third step is to search for objects that need to be added to the worklist. Specifically, the framework checks all of the objects pointed to by obj and sees whether they are already in NVM. If not, they are added to the worklist. It is necessary to move these objects to NVM because they are now reachable from the durable root set. For each of the objects moved to the worklist, steps one to three are repeated, iteratively.

Figure 1 (b) shows the state of the heap after A has been processed. The original A is now a forwarding object to A in NVM, and the latter has the Queued bit set. Also, B is added to the worklist since it is pointed to by A.

Figure 1 (c) shows the final state after the object movement completed. F points to A in NVM, there are two forwarding objects, and all Queued bits are clear. Forwarding objects are only temporary; during garbage collection, this level of indirection is removed and forwarding objects are deallocated.

Finally, the framework also automatically inserts any necessary instructions before and after the write. They may be CLWB and sfence after the write and, if execution is within a transaction, a logging write (plus its CLWB and sfence) before the write.

C. Required Software Checks

A persistence by reachability framework needs to include checks around loads and stores. First, if a persistent object is being updated to point to an object in DRAM, before the write takes place, the actions in Section III-B are performed. Determining whether an object is in the NVM or DRAM heap requires a check on the object's virtual address.

Moreover, before any read or write to an object, one needs to determine whether the object is a forwarding one (i.e., its Forwarding bit is set). If it is, its forwarding pointer must be followed to retrieve the object's true location in NVM. Note that, if the object is in NVM, it cannot be a forwarding one.

Furthermore, before an object can be pointed to by a durable object, one needs to determine whether the object is being processed by another thread to become part of the durable root's transitive closure. This check is required to avoid the inconsistency of a durable object pointing to an object that is not yet part of the durable set. Hence, the software checks if the object is in NVM and has its Queued bit set. If both are true, the object's transitive closure is being processed by another thread, and pointing to the object must be delayed until the Queued bit is cleared.

More explanation can be found in the description of the AutoPersist persistence by reachability NVM framework [21].

IV. P-INSPECT: Main Idea

In persistence by reachability frameworks [21, 22, 23], all the software checks described, plus some runtime decisions based on the results of these checks [27], and the insertion of CLWB, sfence, and logging operations have a large performance impact. We will show in Section IX that they contribute with 22–52% of the instructions in a set of workloads. This is a significant price to pay for the programmability afforded by persistence by reachability. Importantly, most of the checks turn out to require no action. This is because finding objects that are forwarding or whose transitive closure is being moved

to NVM is *not the common case*. In this paper, we propose hardware support to eliminate most of this overhead.

We call our scheme *Persist-Inspect*, or P-INSPECT. It focuses mostly on reducing the overhead of the checks performed before read and write accesses. It also reduces the overhead of the CLWB and sfence instructions that are added to persistent writes. We consider each case in turn.

A. Minimizing the Overheads of Checks

In P-INSPECT, we envision every application read and write to check certain state in hardware. If the check resolves that no special action is needed—recall that this is the common case by far—the read or write completes normally. Otherwise, the hardware automatically invokes a software handler, which performs some checks and extra operations, and also performs the read or write.

To understand the types of checks performed by the hardware, consider the most general operation, where a field in an object (call it *Holder* object) is set to point to another object (call it *Value* object). We represent this as Obj_H.field = Obj_V.

Based on the discussion in Section III, we propose to perform the four checks shown in Table I in hardware, rather than in software as in current systems. The first check is whether the holder and/or the value objects are allocated in NVM or DRAM. If the code is attempting to have an NVM holder object point to a DRAM value object, a software handler will be invoked. The software will copy the value object and its transitive closure to NVM before performing the write.

TABLE I: Hardware checks performed by P-INSPECT.

HW Check	HW Needed
Holder and/or value objects in NVM or DRAM?	Virtual addresses
Holder and/or value objects are forwarding?	Bloom filter
Is the value object's transitive closure being processed?	Bloom filter
Is execution inside a Xaction?	Register bit

The second check is whether the holder and/or the value objects are forwarding objects. If any is, the software will be invoked. It will follow the forwarding link(s) to obtain the correct object(s) before performing the read or write.

The third check is whether the transitive closure of the value object is currently being processed by another thread. Recall from Section III-C that this check is required to avoid the inconsistency of a durable holder object pointing to a value object that is not yet part of the durable set. Depending on the conditions, the software will be invoked. The software will wait until the transitive closure is completely processed before pointing to the value object.

The final check is whether the execution is inside a transaction (Xaction). If so, the software may need to perform a logging operation before the write.

Table I also shows the simple hardware that P-INSPECT uses to perform these checks quickly. Specifically, whether the objects reside in NVM or DRAM can be determined by their virtual addresses. To detect whether an object is a forwarding one, P-INSPECT uses a bloom filter that keeps the addresses of all the current forwarding objects. Similarly, whether the transitive closure of an object is being processed is detected by accessing another bloom filter that keeps the addresses of all such current objects. Finally, whether the execution is inside a Xaction is determined by a register bit that is automatically set and cleared in hardware when a Xaction starts and ends, respectively.

With this hardware support, P-INSPECT minimizes the checking overheads of persistence by reachability. Specifically, write and read instructions automatically trigger the checking hardware described. In the common case when the hardware determines that no special action is needed, the write or read is performed at high speed.

In the uncommon case when the hardware determines that a special action is needed, a software handler is automatically invoked. The handler reads information from the header of the software object structures to determine what actions to take before performing the access— i.e., copy objects, follow forwarding pointers, wait until a transitive closure is fully processed, or log a value inside a Xaction.

B. Minimizing Persistent Write Overheads

A second overhead in practically all NVM frameworks is the fact that performing a persistent write typically requires executing a CLWB and, depending on the persistency model, an sfence (Section II).

P-INSPECT minimizes this overhead by supporting one type of persistent write operation that combines the write, CLWB, and sfence. Such operation interacts efficiently with the cache coherence protocol. It can be automatically invoked when the hardware performs the write, and can also be invoked by the programmer.

V. P-INSPECT DESIGN

A. Bloom Filter Support

As indicated in Section III, *Forwarding* objects are a key component of a high-speed persistence by reachability NVM framework. When an object A needs to be moved from DRAM to NVM, setting-up a forwarding object induces only modest overhead on the critical path of the application. The alternative is: (i) block all the other threads of the application, (ii) traverse the heap to find all the pointers to object A and to its transitive closure [23], and (iii) update these pointers to point to new objects now allocated in NVM. These actions are on the critical path of the application.

Unfortunately, in frameworks with forwarding objects, on an access to an object, one needs to check the Forwarding bit of the object and, if set, follow a pointer to access the correct object in NVM. In addition, on an access to a value object, one needs to check its Queued bit, and only proceed with the access when the bit is clear.

These checks are on the critical path. However, typically, the bits are clear. Hence, P-INSPECT introduces two bloom filters that quickly return whether an object's Forwarding or Queued bits are set. These filters are called Forwarding (*FWD*) and Transitive Closure (*TRANS*), respectively.

FWD Bloom Filter. Immediately before the runtime sets up a forwarding object in DRAM, the runtime inserts the base address of the object in the FWD bloom filter. Later, on a read or write to an object, the hardware searches the FWD filter and determines whether the object is a forwarding one.

When the FWD bloom filter fills-up to a certain threshold, the hardware wakes up a *Pointer Update Thread* (PUT). The PUT traverses all live objects of the volatile heap. When it identifies a pointer to a forwarding object, it updates the pointer to point to the corresponding NVM object. When the PUT has traversed all the objects, it clears the FWD bloom filter in bulk. The now inaccessible forwarding objects will later be reclaimed by the garbage collector.

Section VI discusses the implementation of the FWD filter so that the PUT can operate in the background without stalling program threads or losing any filter information.

TRANS Bloom Filter. Immediately before a value object in the worklist for a transitive closure being processed is moved by the runtime to the NVM, the runtime inserts the base address of the object in the TRANS filter. Later, on an access to a value object, the hardware searches the TRANS filter and determines whether the object is a queued one.

As soon as the thread that is processing the transitive closure sets-up forwarding objects for all the objects in the transitive closure, it clears the TRANS bloom filter in bulk.

B. New Operations

To access the bloom filters, P-INSPECT introduces the new operations shown in Table II. Six of them operate as store instructions and one as a load. They all take at most a memory address and a register as arguments. A possible implementation in x86 could use existing store and load opcodes preceded by a prefix. In the table, *Ha* is the address of a field in a holder object, *Va* is the address of the base of a value object, and *Addr* is the address of the base of an object.

TABLE II: New operations. In the table, *Ha* is the address of a field in a holder object, *Va* is the address of the base of a value object, and *Addr* is the address of the base of an object.

Name	What it Does
checkStoreBoth [Ha],Va	Performs checks, then Mem[Ha] = Va
checkStoreH [Ha],value	Performs checks, then Mem[Ha] = value
checkLoad [Ha],dest	Performs checks, then dest = Mem[Ha]
insertBF$_{FWD}$ Addr	Inserts Addr in the FWD bloom filter
insertBF$_{TRANS}$ Addr	Inserts Addr in the TRANS bloom filter
clearBF$_{FWD}$	Clears the FWD bloom filter
clearBF$_{TRANS}$	Clears the TRANS bloom filter

checkStoreBoth [Ha],Va performs some hardware checks detailed in Section V-C and, if successful, stores the base address of the value object into a field of the holder object. *checkStoreH [Ha],value* also performs hardware checks and, if successful, stores the value into a field of the holder object. *checkLoad [Ha],dest* performs hardware checks and, if successful, loads the contents of a field of the holder object into a destination register. *insertBF$_{FWD}$ Addr* and *insertBF$_{TRANS}$ Addr* insert the base address of an object (Addr) into the FWD and TRANS bloom filter, respectively. *clearBF$_{FWD}$* and *clearBF$_{TRANS}$* clear the FWD and TRANS bloom filter, respectively.

In the first three operations, if the hardware checks are unsuccessful, the write/read is not performed and a software handler is invoked. We now examine the checks and the software handlers.

C. Hardware Checks Before Writes/Reads

Table III shows the hardware checks performed by the *checkStoreBoth*, *checkStoreH* and *checkLoad* operations.

TABLE III: Checks performed by the checkStoreBoth (CSB), checkStoreH (CSH), and checkLoad (CL) operations.

Hardware Check	Operation		
	CSB	CSH	CL
Is Base(Ha) in NVM or DRAM?	✓	✓	✓
Is Va in NVM or DRAM?	✓		
Is Base(Ha) in the FWD bloom filter?	✓	✓	✓
Is Va in the FWD bloom filter?	✓		
Is Va in the TRANS bloom filter?	✓		
Is execution inside a Xaction?	✓	✓	

checkStoreBoth (CSB). This operation checks the most conditions. The first two conditions are whether the base addresses of the two objects accessed (i.e., Base(Ha) and Va) are in NVM or in DRAM. This information is attained by examining the objects' virtual addresses, which tell whether the objects are in NVM or DRAM. The Base(Ha) is directly obtained from the instruction, which contains base plus offset for Ha. The next two conditions are whether the two objects are forwarding objects. This is obtained by hashing Base(Ha) and Va for membership in the FWD bloom filter. The next condition is whether the value object is currently being processed as part of a transitive closure worklist. This information is obtained by hashing Va for membership in the TRANS bloom filter. Finally, the last check is whether the execution is inside a transaction. This is obtained by reading a register bit.

Given all these checks, there are three cases when the hardware can complete the checkStoreBoth operation by performing the write without invoking the software. These three cases are shown in the top three rows of Table IV.

The first row is the case when both objects are in NVM, the value object is not in the TRANS bloom filter, and execution is not inside a Xaction. Since the value object is not in the TRANS bloom filter, there is no need to wait. Further, since execution is not in a Xaction, there is no need to log. Hence, checkStoreBoth performs a read and a persistent write.

The second row is when both objects are in DRAM and not in the FWD bloom filter. They are volatile, non-forwarding objects. checkStoreBoth simply performs a read and a non-persistent write to the holder. There is no need to wait for any transitive closure or perform any logging.

The third row is when the holder is a non-forwarding object in DRAM, and the value object is in NVM. Since having a pointer from DRAM to NVM is always fine, checkStoreBoth simply performs a read and a non-persistent write to the holder.

TABLE IV: Execution flows for stores. Empty boxes (-) can take any values.

Conditions						Action Taken
Where is Base(Ha)?	Base(Ha) in FWD?	Where is Va?	Va in FWD?	Va in TRANS?	In Xaction?	
NVM	-	NVM	-	false	false	HW
DRAM	false	DRAM	false	-	-	HW
DRAM	false	NVM	-	-	-	HW
DRAM	true | -	- | DRAM	- | true	-	-	SW: ①
NVM	-	DRAM | NVM	-	- | true	-	SW: ②
NVM	-	NVM	-	false	true	SW: ③

checkStoreH (CSH). This operation is like checkStoreBoth, except that it does not read from a value object. Instead, it reads from a primitive type like an integer. Hence, checkStoreH does not check anything related to any value object Va (Table III).

As a result, checkStoreH can perform the operation in hardware under the same three cases as checkStoreBoth (Table IV)— except that there is no check for Va conditions. Specifically, checkStoreH performs the read and write and completes if (i) the holder is in NVM and execution is not in a Xaction (first row) or (ii) the holder is a non-forwarding object in DRAM (second and third rows).

checkLoad (CL). Since checkLoad is a read of the holder object, we only need to ensure that the read gets the correct address. Hence the only checks performed are whether the base of Ha is in NVM or DRAM, and whether the object is a forwarding one (Table III). There is no value object and no logging is ever needed.

Given these checks, there are two cases when checkLoad can perform the read and complete. They are the top two rows of Table V. One is when the object is in NVM; the other is when it is in DRAM and is not in the FWD bloom filter.

TABLE V: Execution flows for loads.

Conditions		Action Taken
Where is Base(Ha)?	Base(Ha) in FWD?	
NVM	-	HW
DRAM	false	HW
DRAM	true	SW: ④

D. Software Handlers

In all the other conditions not covered in Section V-C, the hardware does not perform the read or write. Instead, it invokes one of the following 4 software handlers.

Handlers for Writes. As can be seen in Table IV, there are three possible cases when checkStoreBoth or checkStoreH invoke a software handler.

The first case, shown in Row 4, invokes handler ① when: (i) the holder object is in DRAM, and (ii) the value or holder objects are in the FWD bloom filter (i.e., either one or both). In this case, the software needs to check if indeed these objects are forwarding and, if so, follow the forwarding pointer(s) to get to the correct object(s). Recall that a bloom filter can produce false positives (but never false negatives). Hence, to be certain that the object(s) are forwarding ones, the software needs to check the actual Forwarding bits in the headers of the object(s)' software structures.

Further, before performing the write, the software will have to wait until the completion of any in-progress transitive closure processing that includes the value object, and will have to create a log entry if execution is in a Xaction. After that, the software may perform a persistent or a regular write.

We refer to handler ① as *checkHandV*, and show it in Algorithm 1. In Lines 2 and 4, it checks the Forwarding bits of the holder and value objects, respectively, and follows the forwarding links, if needed.

Then, *CheckHandV* determines if the holder object is persistent (i.e., it is in NVM or is Forwarding) (Line 5). If it is not, the algorithm performs a non-persistent write (Line 18). Otherwise, we need to check if the value object is persistent (Line 6) and, if it is not, make it persistent (Line 9). It will not be persistent if either its virtual address is not in NVM or it is part of an in-progress transitive closure processing. The latter condition is indicated with the Queued bit set in the header of V. The *makeRecoverable* function (Line 9) will make it persistent.

After that, if we are in a Xaction, the software creates a log entry and performs a write to NVM. The instruction used, called *persistentWrite*, is discussed in Sec. V-E. This flavor of persistentWrite includes a CLWB but not an sfence, since we are inside a Xaction; we will need an sfence at the end of the Xaction. If we are not in a Xaction, the software performs a write to NVM. In this case, we use a persistentWrite flavor that includes a CLWB and also possibly an sfence.

The second case when a software handler needs to be invoked is shown in Row 5 of Table IV. In this case, we invoke handler ② when: (i) the holder object is in NVM, and (ii) the value object is in DRAM (may or may not be a forwarding object), or in NVM and its Queued bit is set (i.e., it is part of an in-progress transitive closure processing). Only checkStoreBoth can invoke it.

We refer to handler ② as *checkV* in Algorithm 1. The flow is like *CheckHandV* except that no check needs to be performed on the holder object. This is because the object is in NVM.

Finally, the third case when a software handler needs to be invoked is Row 6 of Table IV. This case invokes handler ③ when: (i) both holder and value objects are in NVM, (ii) the value object's Queued bit is clear, and (iii) we are in a Xaction.

Algorithm 1 Software handlers.

```
 1: function ① CHECKHANDV (Ha, Va, Xaction)
 2:     Read H header & update Ha if needed // forwarding case
 3:     if isObject(Va) then
 4:         Read V header & update Va if needed // forwarding case
 5:     if isPersistent(H) then
 6:         if !isPersistent(V) then
 7:             // not in NVM or the Queued bit is set
 8:             // may wait until Queued is cleared
 9:             makeRecoverable(V)
10:         if Xaction then
11:             Write to log // includes a CLWB and sfence
12:             persistentWrite [Ha], Va // persistent program store
13:             // it includes CLWB but not sfence
14:         else
15:             persistentWrite [Ha], Va // persistent program store
16:             // it includes CLWB and possibly also sfence
17:     else
18:         store [Ha],Va // non-persistent program store
19:
20: function ② CHECKV (Ha, Va, Xaction)
21:     Read V header & update Va if needed // forwarding case
22:     if !isPersistent(V) then
23:         // not in NVM or the Queued bit is set
24:         // may wait until Queued is cleared
25:         makeRecoverable(V)
26:     if Xaction then
27:         Write to log // includes a CLWB and sfence
28:         persistentWrite [Ha], Va // persistent program store
29:         // it includes CLWB but not sfence
30:     else
31:         persistentWrite [Ha], Va // persistent program store
32:         // it includes CLWB and possibly also sfence
33:
34: function ③ LOGSTORE (Ha, Va, Xaction)
35:     Write to log // includes a CLWB and sfence
36:     persistentWrite [Ha], Va // persistent program store
37:     // it includes CLWB but not sfence
38:
39: function ④ LOADCHECK (Ha)
40:     Read H header & update Ha if needed // forwarding case
41:     load [Ha]
```

This is a simple case. Handler ③, which we call *logStore*, is shown in Algorithm 1. The handler creates a log entry and performs a store to NVM. Again, this write includes a CLWB but no sfence.

Handlers for Reads. As can be seen in Table V, there is one case when checkLoad needs to invoke a software handler. It invokes handler ④ when the holder object is in DRAM and in the FWD bloom filter. This means that the object may be forwarding. We refer to handler ④ as *loadCheck* in Algorithm 1. The software checks the Forwarding bit in the object's header and, if set, follows the forwarding link (Line 40). Then, it reads the field.

E. Low-Overhead Persistent Write

A write to NVM can be expensive, since it is often followed by a CLWB, and sometimes even by an sfence. Consider the worst case when all three operations need to take place (Figure 2(a)). First, the write itself may have to bring the line from main memory, as it loads it into an L1 cache in Dirty state (Steps ① to ④). The CLWB then needs to find a copy of the line—which is likely to be in the L1 cache but may be in any cache in any state, if the line has been accessed by other cores since the above write. Once the line is found in a cache, it is written to main memory and a copy is kept in that cache (Steps ⑤ to ⑥). Once the acknowledgment of the CLWB completion reaches the originating core (Steps ⑦ to ⑧), the sfence retires, allowing a subsequent write to be merged with the memory system. In the worst case, the combined operations may require two round trips to memory.

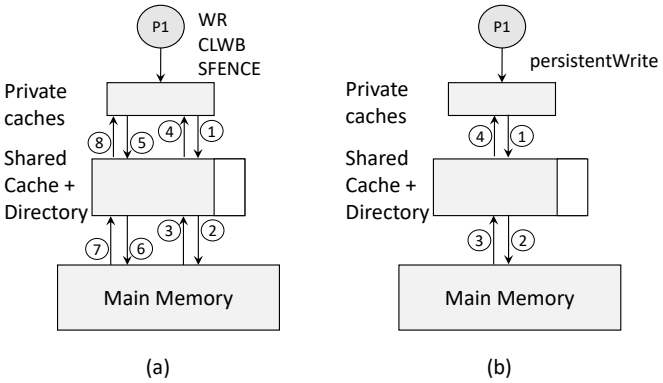

Fig. 2: Conventional persistent write, CLWB, and sfence (a), and proposed advanced persistentWrite flavor (b).

We propose a new instruction called *persistentWrite* that speeds-up a write to NVM. *persistentWrite* has three flavors: one that simply performs a write; one that combines a write with a CLWB; and one that combines a write with a CLWB and an sfence. The flavor used depends on whether the write needs to be followed by CLWB and sfence operations. In our discussion, we assume that the NVM can be written at a granularity finer than a cache line [35, 36, 37, 38, 39, 40].

The first flavor is a simple write. The third flavor is the one that improves performance the most over the state of the art. The three operations (write, CLWB, and sfence) are performed with at most one single round trip to memory (Figure 2(b)). Specifically, the originating core's persistentWrite operation first sends the update down the cache hierarchy (Step ①). If the transaction finds a copy of the line, it is incorporated in the message. Once the message reaches the directory, the directory is locked. If the directory indicates that the line is in state Exclusive/Dirty in another cache, that line is recalled and the owner cache is invalidated. In any case, all cached copies of the line are invalidated (except in the cache of the originating core, if it is there). Finally, the update—combined with the corresponding cache line if the line was dirty in the cache hierarchy—is sent to NVM to persist (Step ②).

The NVM returns an acknowledgment, potentially with the updated line, to the directory (Step ③) and then to the originating core (Step ④). The directory marks that core as having the line in Exclusive state, and is unlocked. Once the core receives the acknowledgment, it allows a subsequent write to proceed. The result is at most a single round trip to memory.

The flavor that combines the write and the CLWB proceeds in a similar manner.

In all cases, persistent writes from different cores are not ordered unless they access the same cache line. In such case, the corresponding directory module serializes and orders them based on arrival order.

Finally, the checkStoreBoth and checkStoreH operations also have the three flavors described. The flavor used depends on whether the write, if it succeeds, needs to be followed by CLWB and sfence operations.

VI. P-INSPECT Implementation Issues

A. Operation of the FWD Bloom Filter

As a program executes, threads insert the base addresses of forwarding objects into the FWD filter. Recall from Section V-A that when the FWD filter fills to a certain threshold, the system wakes up the PUT thread. In the background, PUT updates any pointers to forwarding objects to point to the corresponding NVM objects, and clears the FWD filter.

To enable this operation, P-INSPECT uses two FWD bloom filters—call them red and black. They have one extra bit called *Active*, which is set for the single FWD filter that is currently being inserted to.

Suppose that the red FWD filter is currently the active one. When it fills up to the threshold, PUT wakes up and toggles the Active bit in both FWD filters. PUT then performs a sweep of the objects in the volatile heap, processing encountered pointers to forwarding objects as discussed above. During this time, since the black FWD filter is now the active one, all object insertions required by the program are performed on the black FWD filter. However, object lookups are performed on *both* FWD filters. When PUT finishes its traversal, it clears the red FWD filter and goes back to sleep.

With this approach, PUT execution happens in the background, without stalling program threads, and no filter information is ever lost. It is likely that PUT's execution also updates pointers to some forwarding objects that were inserted in the black FWD filter. Hence, the black FWD filter may now include some objects that are not forwarding anymore. This is no problem since, at worse, this effect increases the number of false positives in the FWD bloom filter.

B. Implementing the Bloom Filters

Each process has two FWD bloom filters, each with 2047 bits for the data and 1 bit (the most significant one) for the Active bit. Hence, a FWD filter covers 4 cache lines in a machine with 64-byte cache lines. The TRANS bloom filter only uses 512 bits, which is 1 cache line. Overall, the bloom filters for a process use 9 cache lines. For each filter, we use two hash functions, H_0 and H_1. Each process has all of its bloom filters in memory in a single page, at a fixed virtual address. The filters are accessible with virtual addresses.

The operations performed on the FWD bloom filters are shown in Table VI. The operations on the TRANS filter are fewer and simpler; we do not show them for simplicity. In an *Object Lookup*, we take the object's address and hash it using H_0 and H_1. Then, we read the 2 FWD bloom filters and check for membership in them.

In an *Object Insert*, we hash the object's address using H_0 and H_1 as before. Then, we read the most-significant cache line of the two filters and identify which filter is the active one. Finally, we read the three remaining lines of the active filter and set the appropriate bits in the filter.

TABLE VI: Operations performed on the FWD filters.

Operation	What it Does
Object Lookup	Check both FWD filters for address membership
Object Insert	Insert address in the active FWD filter
Inactive FWD Filter Clear	Zero-out the inactive FWD filter
Change Active FWD Filter	Toggle the active bit in both FWD filters

In an *Inactive FWD Filter Clear*, we read the most-significant line of the two filters and identify the inactive filter. Then, we read the three remaining lines of the inactive filter and clear the filter. Finally, in a *Change Active FWD Filter* operation, we read the most-significant line of the two filters, and toggle the active bit in both.

P-INSPECT implements these operations in hardware. Specifically, P-INSPECT has a *BFilter_FU* functional unit that helps execute the instructions of Table II. This functional unit knows the address and layout of the page of bloom filters, and the hash functions H_0 and H_1. Given an address, it can perform H_0 and H_1 on it, and access the filters.

The left part of Figure 3 shows the P-INSPECT hardware in the core. In addition to the *BFilter_FU* functional unit with the two hash functions, there is a bit that indicates if execution is inside a transaction, and the virtual addresses for: the bloom filter page, the software handlers, and the base and limit of the persistent heap.

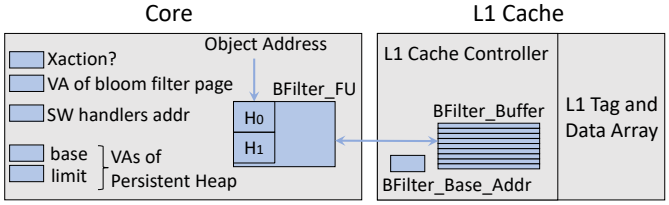

Fig. 3: Implementation of the P-INSPECT architecture.

The operations in Table VI are implemented as follows. An Object Lookup is performed in hardware, as part of the checkStoreBoth, checkStoreH, and checkLoad operations. The lookup uses the BFilter_FU functional unit, and is fully overlapped with the store or load.

An Object Insert is performed by the runtime executing the $\text{insertBF}_{\text{FWD}}$ or $\text{insertBF}_{\text{TRANS}}$ operations in Table II for the FWD or TRANS filter, respectively. An Inactive FWD Filter Clear is performed by the PUT thread executing the $\text{clearBF}_{\text{FWD}}$ operation in Table II when it has completed its operations on the volatile heap. Similarly, the TRANS filter is cleared by a thread executing the $\text{clearBF}_{\text{TRANS}}$ operation in Table II when it has completed processing a transitive closure. Finally, the Change Active FWD Filter operation is performed by the PUT thread when it wakes up.

C. Keeping Bloom Filter Data Coherent

In a multithreaded program running on a multiprocessor, we need to ensure the coherence of the bloom filter data across cores. To understand the problem, recall that while Object Lookup is a read-only operation, the other three operations (Object Insert, Inactive FWD Filter Clear, and Change Active FWD Filter) are read-write operations that need to be performed atomically. Note that insertBF_{TRANS} and clearBF_{TRANS} also include write operations. However, as we will see, the read-only Object Lookup operation is on average over one million times more frequent than the operations that involve writes. We want to keep the former fast.

To keep bloom filter data coherent, a simple approach is to use the protection bits in the TLB for the bloom filter page. Specifically, when a thread wants to perform a read-write operation, it sets the owner bit in its TLB entry and invalidates the TLB entry from the TLBs of the other cores. Unfortunately, this approach is too slow, as it involves the OS.

A faster approach, used by P-INSPECT, is to use the cache coherence protocol to maintain the coherence of bloom filter data. To understand the approach, assume a single bloom filter that spans a single cache line. When performing an object lookup, the hardware requests the cache line in Shared state. When performing the other operations, the hardware requests the line in Exclusive state. In this case, when the cache obtains the line, the cache refuses any incoming transaction to the line (i.e., it locks it) until all the local reads and writes to the bloom filter are done. Then, it unlocks the line.

A difficulty with this approach is that the bloom filters in P-INSPECT span 9 contiguous cache lines. In effect, we would like the coherence protocol to operate on these lines as if they were "glued" together—i.e., all 9 lines should be fetched at a time, and all 9 kept in the cache at a time.

To solve this problem, P-INSPECT augments the L1 cache controller with a register that contains the address of the bloom filter lines used by the currently-executing process. Since these 9 lines are contiguous, the register only keeps the base address. We call it the *BFilter_Base_Addr* register. In addition, the controller has a small buffer that has space for the 9 lines of bloom filter data. We call this buffer the *BFilter_Buffer*, and its lines are visible to the cache coherence protocol. These components are shown in the right part of Figure 3.

In an Object Lookup operation, as the BFilter_FU functional unit requests the bloom filter lines, the cache controller attempts to read all 9 lines into the BFilter_Buffer in Shared state. If, as it reads lines, some get invalidated by writes from other cores, it retries the read. When all 9 lines are obtained, they are read by the BFilter_FU.

In the bloom-filter operations that involve read-write accesses, we use the most significant line of the red FWD bloom filter as the *Seed*. This means that, as the BFilter_FU functional unit requests the bloom filter lines in Exclusive state, the cache controller attempts to obtain the Seed cache line in Exclusive state *first*. Once it attains it, it locks it in the BFilter_Buffer and proceeds to obtain the remaining lines in the BFilter_Buffer in Exclusive state. These lines are also locked. Once all the lines are present, they are read by the BFilter_FU, which performs the read-write operations. After that, the lines are unlocked and accept external requests. Since obtaining the Seed cache line in Exclusive state serializes all the other operations, there is no data incoherence or deadlock.

With this design, at a context switch, the OS simply writes back the dirty lines in the BFilter_Buffer, invalidates the BFilter_Buffer lines, and updates the BFilter_Base_Addr register with the base address of the bloom filters for the new process. The lines with the bloom filter data of the new process will be brought into the BFilter_Buffer on demand.

VII. Relation to Failure Recovery

If an NVM system is to recover from failures, it needs software that performs operations such as undo/redo logging or checkpointing. Such software has to be aware of the memory persistency model [41] used by the system, since the memory persistency model determines when, after persistent objects are written, will the updates reach NVM.

A persistence by reachability framework such as P-INSPECT does not impact failure recovery. The framework's goal is to simplify the programmer's job by performing two main tasks automatically: (i) ensure that objects that need to be persistent are moved from volatile memory to NVM at the right time, and (ii) ensure that the updates to such objects are marked as being persistent—and therefore are accompanied by the correct CLWB and sfence instructions. The actual CLWB and sfence instructions added with the updates depend on the memory persistency model used by the system. Hence, the persistence by reachability framework is cognizant of the persistency model used; at no point, however, it affects or needs to know about the failure recovery algorithms.

A related issue is that prior literature has proposed per-core buffers that buffer stores destined for NVM [42]. Then, there is hardware that optimizes the write back of these stores to NVM to improve performance. One concern in this environment is to honor all inter-thread persistence dependencies. Once again, these hardware optimizations are orthogonal to the persistence by reachability framework. The latter simply ensures that the updates to the buffers are performed transparently to the programmer, efficiently, and following the memory persistency model used.

VIII. Evaluation Methodology

Modeled Architecture and Infrastructure. We use cycle-level simulations to model a server architecture with 8 cores and a main memory of 32 GBs of NVM and 32 GBs of DRAM. The main architecture parameters are shown in Table VII. We integrate the Simics full-system simulator [43] with the SST [44] framework and the DRAMSim2 [45] memory simulator. To model NVM, we modified the DRAMSim2 timing parameters as shown in Table VII, and disabled refreshes. We use Intel SAE [46] on top of Simics for OS instrumentation, the Synopsys Design Compiler [47] to evaluate the RTL implementation of CRC hash functions, and

CACTI [48] for the area and energy analysis of the hardware structures at 22nm.

For the runtime system, we use the most recent version of the AutoPersist framework [21]. AutoPersist is built within the Maxine Java Virtual Machine (JVM) [49]. In addition, to perform a lengthy analysis of the behavior of the architecture, we use Pin [50]. We augment the Java compiler and runtime to communicate the required information to our simulation infrastructure, both to Simics and Pin.

TABLE VII: Architectural parameters used for evaluation.

Processor Parameters	
Multicore chip	8 OoO cores, 2GHz, 2 issue (and 4 issue)
Ld-St queue; ROB	92 entries; 192 entries
Cache Line Size	64 bytes
DL1 cache	32KB, 8-way, 2-cycle access latency
L2 cache	256KB, 8-way, 8-cycle data, 2-cycle tag lat.
L3 cache	1MB/core, 16-way, 22-cycle data, 4-cycle tag latency
Cache coherence	MESI protocol
L1 TLB	64 entries, 4-way, 2-cycle latency
L2 TLB	1024 entries, 12-way, 10-cycle latency
Bloom Filter Parameters and Analysis	
Size	FWD: 2047 bits; TRANS: 512 bits
Hash function	CRC; 2-cycle latency; Area: $1.9 * 10^{-3} mm^2$; Dyn. energy: $0.98pJ$; Leak. power: $0.1mW$
Call to PUT	When 30% of FWD bloom filter bits are set
BFilter_Buffer	Area: $0.023mm^2$; Leakage power: $1.9mW$; Rd/Wr energy per access: $12.8/13.1pJ$ Lookup access overlaps with ld/st (2 cycles)
Main-Memory Parameters	
Channels; Banks	DRAM: 2; 8 NVM: 2; 8
t_{CAS}-t_{RCD}-t_{RAS}	DRAM: 11-11-28; NVM: 11-58-80
t_{RP},t_{WR}	DRAM: 11,12 NVM: 11,180
Freq; Bus width	1GHz DDR; 64 bits per channel
Host and Runtime System Parameters	
Host OS; Runtime	Ubuntu Server 16.04; Maxine JVM 2.0.5

Configurations. We compare four different designs.

Baseline: It uses the unmodified AutoPersist [21], a Java programming framework that provides persistence by reachability. AutoPersist performs all the runtime checks and object moves.

P-INSPECT--*:* AutoPersist plus our proposed P-INSPECT hardware to perform the required checks for loads and stores, but does not include the optimization for speeding-up persistent writes (Section V-E).

P-INSPECT*:* AutoPersist plus the complete P-INSPECT design, including the optimization for persistent writes.

Ideal-R: AutoPersist without all the checks and object moves required to implement persistence by reachability. It is an ideal runtime where the user identified all persistent objects. It does not include the persistent write optimization.

Workloads. We do experiments on a key-value store using different backends, and also on several kernel applications.

Key-Value Store: We implement a persistent version of a key-value store by modifying QuickCached [51] to use the AutoPersist framework to persist its internal key-values. For our evaluation, we use four different, commonly used backends: (i) ***pTree*** uses a Java implementation of the IntelKV B+ tree [52] and persists both all inner and leaf nodes; (ii) ***HpTree*** uses the same data structure as pTree, but it is a hybrid design that only persists the leaf nodes of the tree, similar to IntelKV; (iii) ***hashmap*** uses a HashMap data structure for its internal storage; and (iv) ***pmap*** uses the Map implementation from the Java PCollections library [53].

To evaluate the performance of our key-value stores we use the Yahoo! Cloud Serving Benchmark (YCSB) [54]. This benchmark suite is commonly used for the evaluation of cloud storage services. We populate our key-value stores to a memory footprint of 12.5GB for pTree and HpTree, 12.4GB for hashmap, and 12.8GB for pmap. In our evaluation, we run three of the YCSB workloads: (i) the write-intensive workload A, (ii) the read-intensive workload B, and (iii) workload D, which is both read intensive and also inserts new values into the data structures.

Kernel Applications: These are several kernels that perform a collection of read, write, insert, and delete operations on persistent data structures. In total, we use six different kernel applications: (i) ***ArrayList*** is a persistent version of the Java ArrayList; (ii) ***ArrayListX*** is identical to the previous kernel, but uses transactions to perform in-place insertions and deletions; (iii) ***LinkedList*** is a doubly linked list implementation; (iv) ***HashMap*** is a HashMap implementation; (v) ***BTree*** is a B-tree implementation; and (vi) ***BPlusTree*** is a B+ tree implementation. We populate the kernel applications with one million elements before simulation.

Simulation Methodology. We perform two types of simulations. One is architectural simulations to evaluate performance using a cycle-level simulator. The other is behavioral simulations using Pin to characterize the behavior of applications and bloom filters over long execution intervals.

For the detailed architectural simulations, we perform full-system simulations. We warm-up the architectural state by running, per core, 200M instructions before simulating 1B instructions. For the behavioral simulations, we run hundreds of billions of instructions with Pin.

IX. Evaluation

A. Architectural Evaluation

The goal of P-INSPECT is to minimize the overheads of programmable NVM frameworks. P-INSPECT targets two main sources of overhead in such frameworks: (i) the runtime checks that need to be performed on the objects to determine their state, and (ii) the cost of the persistent write operations. In this section, we consider these overheads.

Figure 4 shows the instruction count of the kernels for the different configurations. The count is normalized to that of the baseline configuration. We see that P-INSPECT-- and P-INSPECT greatly reduce the number of instructions executed across the board. On average, they reduce the number of instructions by 46%. Kernels that have a larger number of stores like ArrayList attain higher instruction reductions than those that are more read-intensive like BTree. Recall that stores require more checks.

The figure also shows that P-INSPECT-- and P-INSPECT have approximately the same instruction count. Further, they

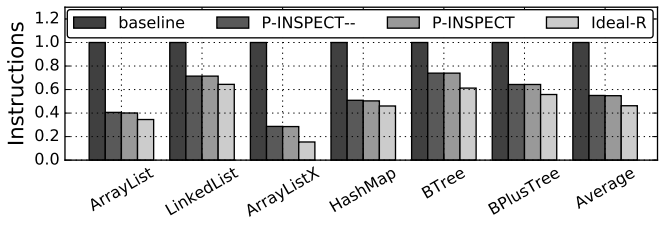

Fig. 4: Instruction count of the kernel applications.

are fairly close to the Ideal-R configuration, which achieves an average 54% reduction.

Figure 5 shows the total execution time of the kernels. The figure is organized as Figure 4, except that we have broken down the baseline bar into time to perform: (i) the checks (baseline.ck), (ii) the persistent writes, approximately (baseline.wr), (iii) the persistency by reachability runtime operations such as logging and copying objects between DRAM and NVM (baseline.rn), and (iv) the rest (baseline.op). Baseline.op corresponds to a true ideal system with no persistency by reachability and (unlike Ideal-R) no NVM.

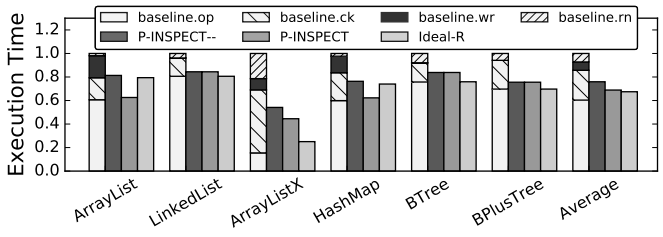

Fig. 5: Execution time of the kernel applications.

We see that, in the baseline, the checking overhead is substantial, while the persistent write overhead is sometimes significant, and the runtime overhead is only significant when there is logging (ArrayListX). Both P-INSPECT-- and P-INSPECT deliver significant speed-ups. On average, they are 24% and 32% faster than baseline, respectively. The speed-ups, of course, are smaller than the instruction count reductions. However, the trends are largely similar.

We see a difference between P-INSPECT-- and P-INSPECT for some applications. Although these two configurations have similar instruction counts in Figure 4, P-INSPECT has a lower execution time for applications that have many persistent writes, like ArrayList and HashMap. This is especially the case when these writes miss in the cache hierarchy. In this case, thanks to our design, combining a write with a CLWB and an sfence has a substantial performance impact.

Finally, the average speed-up of P-INSPECT is very close to that of Ideal-R, which reduces execution time by 33%. In some cases, P-INSPECT is even faster than Ideal-R. The reason is that P-INSPECT includes the persistent write optimization of Section V-E, while Ideal-R does not.

Figures 6 and 7 show the instruction count and the execution time, respectively, for the key-value stores. The figures are organized as the previous ones. In general, the trends for instruction reduction and execution time reduction are like for kernels, except that the improvements are relatively smaller. This is because these workloads perform relatively more non-memory access instructions than the kernels.

From Figure 6, we see that, on average, P-INSPECT reduces the executed instructions by 26%. The same reduction is obtained by P-INSPECT--. This reduction is close to the 31% reduction attained by Ideal-R. The instruction reduction is larger in the write-heavy workload A than in the other workloads. This is because writes are more check-intensive. In hashmap-A the reduction reaches 50%.

Figure 7 shows that, on average, P-INSPECT-- and P-INSPECT reduce the execution-time of the key-value stores by 14% and 16%, respectively, relative to baseline. These are substantial reductions for real-world workloads. Note that Ideal-R delivers a 17% reduction in execution time, which is only 1 percentage point more than P-INSPECT. Hence, the P-INSPECT hardware effectively hides the overhead of persistence by reachability. Further, for some persistent write intensive workloads such as hashmap-A, P-INSPECT is faster than Ideal-R.

Figure 7 breaks down the execution time of the applications in baseline like in Figure 5. We see that, as in Figure 5, the checking overhead is the most dominant one.

To get another insight into the benefit of accelerating persistent writes, we have isolated the persistent writes within all the applications, and added-up the total time it takes for them to complete (i.e., we do not consider any overlapping with other instructions). We compare the time it takes to execute the separate write, CLWB, and sfence instructions against when the instructions are combined in a *persistentWrite* P-INSPECT operation. This metric ignores any overlap that these instructions may have with any other instruction.

It can be shown that our *persistentWrite* operations that combine writes, CLWBs, and sfences take, on average, 15% less time than the instructions separated. In fact, for ArrayList, the reduction is 41%.

B. Bloom Filter Evaluation

We now characterize the behavior of P-INSPECT and its bloom filters over long execution runs using Pin. The TRANS bloom filter is cleared often, when the processing of a transitive closure completes. Because of that, we find that the TRANS bloom filter has a false positive rate close to zero. Hence, we do not consider it further.

The FWD bloom filter is cleared less frequently, by the PUT thread. PUT is woken up when FWD has over 30% of its bits set. We find that, on average, 357 forwarding objects are inserted in FWD before this threshold is reached. Our experiments also show that the average false positive rate of FWD across all the benchmarks is 2.7%. However, the actual rate of calling a software handler because of a false positive is less than 1%. This is because, in many scenarios, the outcome of the FWD check does not determine whether to call a software handler (Tables IV and V).

To characterize FWD, we run all the applications with the same configuration parameters as before, but with the ratio of operations of the YCSB workloadd: 5% of inserts and 95%

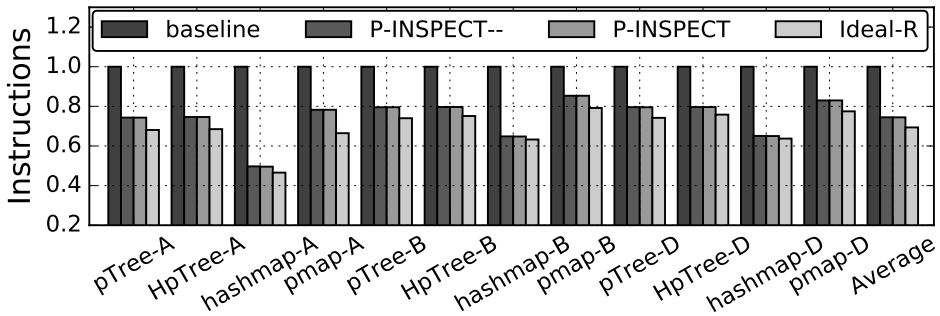

Fig. 6: Instruction count of the YCSB workloads.

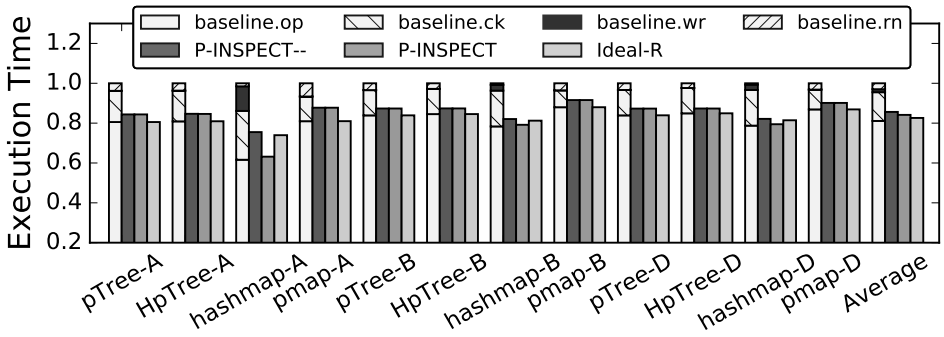

Fig. 7: Execution time of the YCSB workloads.

reads. We collect 50 samples per application and report the mean. Table VIII shows the results for each application.

TABLE VIII: Characterization of the FWD bloom filter.

Applic.	# Inst. between PUT calls (mill.)	# FWD checks per insert (thous.)	Avg. FWD occup.	PUT instr.
ArrayList	26,326	3,006.0	14.5%	0.0%
LinkedList	3,175	163.5	15.9%	0.2%
ArrayListX	43,778	4,937.4	15.8%	0.0%
HashMap	928	134.8	15.9%	1.6%
BTree	237	10.4	15.9%	6.5%
BPlusTree	45,367	3,201.0	15.9%	0.0%
pTree-D	478	22.4	16.0%	3.6%
HpTree-D	426	11.1	15.8%	3.8%
hashmap-D	969	85.2	16.1%	1.8%
pmap-D	92	1.9	15.9%	18.4%
Average	12,177	1,157.4	15.8%	3.6%

Column 2 shows the number of instructions executed between invocations of the PUT. We see that this number ranges from 92 million to 45 billion. Generally, PUT is invoked rarely. Column 3 shows the number of FWD checks divided by the number of FWD insertions. We see that FWD reads are much more frequent than writes: on average, we have 1.15 million reads per write. Column 4 shows the average FWD occupancy. We take a sample every time that the program performs a FWD lookup. We see that this number is low. Its range is 14-16%. Column 5 shows the additional instructions executed by the PUT relative to the application instructions. On average, we see that the PUT overhead is very small.

The net effect of these measurements is that the PUT does not need to be frequently invoked for the FWD to exhibit a low false positive rate. Moreover, the results show that bloom filters are a low cost hardware mechanism that is very effective to handle object checks accurately.

Finally, we perform a sensitivity analysis of the size of FWD. The goal is to show how the FWD size affects the frequency of calling the PUT. Figure 8 shows the normalized number of instructions between PUT invocations for FWD sizes ranging from 511 bits to 4095 bits. We use the same target occupancy as before. Instruction counts for each application are normalized to the 2047-bit case. The numbers on the bars are the % increase in instruction count due to PUT.

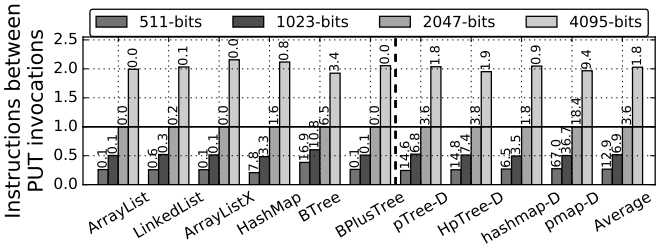

Fig. 8: Normalized number of instructions between PUT invocations for different FWD sizes. The numbers on the bars are the % increase in instruction count due to PUT.

We see there is an almost linear relationship between the FWD size and the frequency of PUT invocations to clear the FWD. There is a tradeoff between FWD size and frequency of PUT invocation. Our 2047-bit design is a good design point.

C. Other Evaluation

To gain more insight into P-INSPECT, we perform two additional experiments. In the first one, we measure, for each application, the percentage of accesses to NVM addresses and the reduction in execution time of P-INSPECT over baseline. Table IX shows the results. We see that both metrics are broadly correlated. There are some cases, however, where the execution time reductions are higher than expected from the fraction of NVM accesses. This effect is due to a higher fraction of persistent writes that miss in the caches, and benefit from our persistentWrite optimization.

TABLE IX: Application NVM accesses and reduction in execution time.

Application	NVM accesses	Execution Time Reduction
ArrayList	13.3%	37.4%
LinkedList	6.4%	15.6%
ArrayListX	14.8%	55.9%
HashMap	8.3%	37.7%
BTree	6.3%	16.2%
BPlusTree	11.3%	24.4%
pTree-D	6.1%	12.8%
HpTree-D	2.8%	12.7%
hashmap-D	7.2%	20.5%
pmap-D	1.0%	9.9%

In a second experiment, we re-run our evaluation with 4-issue (rather than 2-issue) cores. The average speed-ups of P-INSPECT−−, P-INSPECT and Ideal-R over baseline are 23%, 31% and 33% for the kernels, and 14%, 16% and 17% for the workloads, respectively. These numbers are practically the same as for 2-issue. The reason is twofold. First, all the environments become faster (including baseline and P-INSPECT); second, the long-latency NVM accesses stall the pipeline for both issue width designs.

X. Related Work

System Support for NVM. The systems and storage communities have proposed many systems and frameworks to assist NVM integration. Besides the programming frameworks of Section II, researchers have proposed to redesign the software systems. Examples include BPFS [12], NOVA [55, 56, 57, 58], Aerie [59] and PMFS [60]. These systems have different ordering and consistency attributes, and they handle their metadata information differently, but all aim for high performance NVM accesses in hybrid memory.

Hardware Optimizations for NVM. Many works provide hardware optimizations for NVM operations. In general, their goal is to reduce the overhead of persistent writes, either by introducing new persistency models [41, 18], or by removing persistent writes from critical paths and minimizing logging overheads [61, 62, 63, 64, 65, 66, 15, 67, 42, 68, 69]. As NVM normally relies on transactions for memory persistency, most of the optimizations focus on reducing the persistency overhead by relaxing the persistence order. For instance, WHISPER [15] proposes high-level ISA primitives to decouple ordering from durability. Other works propose hardware to optimize different aspects of NVM memories: efficient checkpointing [70, 71], improved NVM encryption operations [72, 73, 74, 75], and persistent object translation to accelerate the process of identifying the addresses of persistent objects [76, 77]. There is no work that proposes hardware techniques for supporting programmable NVM frameworks per se. These proposals are orthogonal to our work and, in fact, many can be combined with our work.

Recognizing Object State. Persistence by reachability requires dynamic fine-grain state information about each individual object. Existing hardware cannot provide the information needed or as fast as it is needed. For instance, bounds checking hardware [28] cannot be used to find out conditions such as whether an object is a Forwarding one or whether its Transitive Closure is being processed (Table I). On the other hand, ARM's Memory Tagging Extension (MTE) [29], SPARC's Application Data Integrity (ADI) [78] or CHERI [30, 31] could be used for fine-grain identification. These proposals tag memory locations with bits that identify their state. However, these approaches are too slow for production code. As documented in [30, 79], MTE's, ADI's, and CHERI's precise exception mode introduces significant performance overheads. Since the hardware first needs to fetch the state (tag or capabilities) of the memory location and check if a precise exception needs to be raised, the original operation can only be performed after this load and check. In P-INSPECT, this overhead does not exist. By using bloom-filter hardware checks, P-INSPECT removes the loading of state from the execution's critical path.

XI. Conclusion

To attain both user-friendly and high-performance NVM frameworks, this paper introduced P-INSPECT, the first hardware architecture targeted to speeding-up persistence by reachability. P-INSPECT retains programmability and eliminates most of the execution overhead. For a set of workloads, it reduces the number of instructions executed by 26%, and the application execution time by 16%, delivering similar performance to that of an ideal runtime.

References

[1] Intel, "3D XPoint: A Breakthrough in Non-Volatile Memory Technology," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2018.

[2] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems*, 1st ed. Morgan & Claypool Publishers, 2011.

[3] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, July 2008.

[4] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, 2009.

[5] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[6] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[7] Intel, "Introduction to Programming with Intel Optane DC Persistent Memory," 2018. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel

[8] T. Shull, J. Huang, and J. Torrellas, "Defining a High-level Programming Model for Emerging NVRAM Technologies," in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ser. ManLang '18. New York, NY, USA: ACM, 2018, pp. 11:1–11:7. [Online]. Available: http://doi.acm.org/10.1145/3237009.3237027

[9] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "Espresso: Brewing Java For More Non-volatility with Non-volatile Memory," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, New York, NY, USA, 2018.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 105–118. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950380

[11] 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. New York, NY, USA: ACM, 2011, pp. 91–104. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[12] 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 *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[13] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 433–452. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660224

[14] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical Persistence for Multi-threaded Applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 468–482. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064204

[15] 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, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, 2017, pp. 135–148.

[16] J. E. Denny, S. Lee, and J. S. Vetter, "NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 125–136. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907303

[17] N. Cohen, D. T. Aksun, and J. R. Larus, "Object-oriented recovery for non-volatile memory," *PACMPL*, vol. 2, no. OOPSLA, pp. 153:1–153:22, 2018.

[18] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level Persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 481–493. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080229

[19] H.-J. Boehm and D. R. Chakrabarti, "Persistence Programming Models for Non-volatile Memory," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA: ACM, 2016, pp. 55–67. [Online]. Available: http://doi.acm.org/10.1145/2926697.2926704

[20] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for Synchronization-free Regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 46–61. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192367

[21] T. Shull, J. Huang, and J. Torrellas, "AutoPersist: An Easy-to-use Java NVM Framework Based on Reachability," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, 2019, pp. 316–332.

[22] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software Persistent Memory," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342850

[23] L. Marmol, M. Chowdhury, and R. Rangaswami, "LibPM: Simplifying Application Usage of Persistent Memory," *ACM Trans. Storage*, vol. 14, no. 4, pp. 34:1–34:18, Dec. 2018. [Online]. Available: http://doi.acm.org/10.1145/3278141

[24] "NVM Programming Model v1.2." [Online]. Available: https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf

[25] "Persistent Memory Development Kit." [Online]. Available: http://pmem.io/pmdk/

[26] "Intel 64 and IA-32 Architectures Software Develop's Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2015.

[27] T. Shull, J. Huang, and J. Torrellas, "QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: ACM, 2019, pp. 137–151. [Online]. Available: http://doi.acm.org/10.1145/3313808.3313822

[28] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3224423

[29] ARM, "Arm Architecture Reference Manual," https://developer.arm.com/docs/ddi0487/latest.

[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," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 457–468, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2678373.2665740

[31] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," *2015 IEEE Symposium on Security and Privacy*, pp. 20–37, 2015.

[32] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for Non-Volatile Main Memory Is Hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*, September 2017, pp. 13:01–13:08.

[33] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 411–425. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304015

[34] "PMDK. An introduction to pmemcheck." https://pmem.io/2015/07/17/pmemcheck-basic.html, 2015.

[35] Zhou, Ping and Zhao, Bo and Yang, Jun and Zhang, Youtao, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1423. [Online]. Available: https://doi.org/10.1145/1555754.1555759

[36] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 347–357.

[37] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, C. Li, G. Xu, and Y. Chen, "Adaptive Granularity Encoding For Energy-Efficient Non-Volatile Main

Memory," in *2019 56th ACM/IEEE Design Automation Conference*, 2019, pp. 1–6.

[38] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *2012 39th Annual International Symposium on Computer Architecture*, 2012, pp. 548–560.

[39] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and endurance-aware design of phase change memory caches," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 136–141.

[40] Y. Xie, "Modeling, Architecture, and Applications for Emerging Memory Technologies," *IEEE Design Test of Computers*, vol. 28, no. 1, pp. 44–51, 2011.

[41] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 265–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712

[42] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 58:1–58:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195709

[43] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[44] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, Mar. 2011.

[45] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, Jan 2011.

[46] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *the 2016 International Conference on Supercomputing*, 2016.

[47] "Synopsys design compiler," https://www.synopsys.com/.

[48] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3085572

[49] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An Approachable Virtual Machine for, and in, Java," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 30:1–30:24, Jan. 2013.

[50] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[51] "QuickCached." [Online]. Available: https://github.com/QuickServerLab/QuickCached

[52] "Pmemkv: Key/Value Datastore for Persistent Memory." [Online]. Available: https://github.com/pmem/pmemkv

[53] "PCollections." [Online]. Available: https://pcollections.org/

[54] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[55] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 323–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930583.2930608

[56] 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. New York, NY, USA: ACM, 2017, pp. 478–496. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132761

[57] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 221–234. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/yang

[58] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 207–219. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/zheng

[59] 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*, ser. EuroSys '14. New York, NY, USA: ACM, 2014, pp. 14:1–14:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592810

[60] 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. New York, NY, USA: ACM, 2014, pp. 15:1–15:15. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592814

[61] 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 International Symposium on Microarchitecture (MICRO-46)*, Davis, CA, 2013.

[62] 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, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, 2017, pp. 178–190.

[63] T. M. Nguyen and D. Wentzlaff, "PiCL: a software-transparent, persistent cache log for nonvolatile main memory," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 178–190.

[64] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 178–190.

[65] M. Ogleari, E. L. Miller, and J. Zhao, "Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, 2018, pp. 336–349.

[66] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-Atomic Persistent Memory Updates via JUSTDO Logging," in *Proceedings of 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, 2016.

[67] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 361–372.

[68] K. Doshi, E. Giles, and P. J. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 77–89.

[69] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building Durable Transactions with Decoupling for Persistent Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 329–343. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037714

[70] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 672–685. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830802

[71] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory," in *2017*

26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sep. 2017, pp. 318–329.

[72] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing Memory and Storage Support for Non-volatile Memory Systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 143–156. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322206

[73] K. A. Zubair and A. Awad, "Anubis: Ultra-low Overhead and Recovery Time for Secure Non-volatile Memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322252

[74] M. Ye, C. Hughes, and A. Awad, "Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 403–415.

[75] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694387

[76] 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*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 800–812. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123981

[77] T. Wang, S. Sambasivam, and J. Tuck, "Hardware Supported Permission Checks on Persistent Objects for Performance and Programmability," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 466–478.

[78] Oracle, "Using Application Data Integrity (ADI)." [Online]. Available: https://docs.oracle.com/cd/E53394_01/html/E54815/gqajs.html

[79] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory Tagging and how it improves C/C++ memory safety," *CoRR*, vol. abs/1802.09517, 2018. [Online]. Available: https://arxiv.org/abs/1802.09517