Defining a High-Level Programming Model for Emerging NVRAM Technologies

Thomas Shull
University of Illinois at Urbana-Champaign
shull1@illinois.edu

Jian Huang
University of Illinois at Urbana-Champaign
jianh@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

ABSTRACT

Byte-addressable non-volatile memory is poised to become prevalent in the near future. Thanks to device-level technological advances, hybrid systems of traditional dynamic random-access memory (DRAM) coupled with non-volatile random-access memory (NVRAM) are already present and are expected to be commonplace soon. NVRAM offers orders of magnitude performance improvements over existing storage devices. Due to NVRAM's low overheads, many future applications are expected to leverage the fine-grain durable storage provided by NVRAM.

Many frameworks for programming NVRAM have been proposed. Unfortunately, these existing frameworks closely mirror the underlying hardware. This lack of abstraction hurts programmer productivity, makes it easy to write buggy code, and limits the compiler's effectiveness. Furthermore, this low level of abstraction does not match the expectations of managed language users.

To rectify this situation, in this paper we describe a new high-level NVRAM programming model amenable to managed languages. Because our model is defined at a high level, it is intuitive, not prone to user bugs, and is flexible enough to allow language implementers to perform many optimizations while still adhering to the model.

In addition to proposing this model, we also briefly describe how Java can be extended to support our new model. Finally, we present some initial results on the performance overheads of creating durable applications in NVRAM and describe what future work we intend to complete.

CCS CONCEPTS

• **Software and its engineering → Data types and structures**; **Source code generation**;

KEYWORDS

Java, Non-Volatile Memory, Programming Model

ACM Reference Format:
Thomas Shull, Jian Huang, and Josep Torrellas. 2018. Defining a High-Level Programming Model for Emerging NVRAM Technologies. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3237009.3237027

ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237027

1 INTRODUCTION

In recent years, technological advances have been made towards having byte-addressable non-volatile memory. Whereas traditionally, durably storing data required the use of block-based storage devices such as Hard Disk Drives (HDDs) or Solid State Drives (SSDs), new device technologies such as Phase-change memory (PCM) [23, 33] and Resistive RAM (ReRAM) [5] that offer non-volatile memory with byte-level access granularily are being rapidly developed. These new technologies are known collectively as non-volatile random-access memory (NVRAM). NVRAM offers substantial performance improvements over traditional storage devices; it has performance similar to current volatile dynamic random-access memory (DRAM), has higher capacities, and retains its values across system restarts. Hybrid systems consisting of both NVRAM and DRAM are imminent; Intel has already released NVRAM products [2] and plans to release many more higher-performing NVRAM products in the second half of 2018.

This introduction of NVRAM with performance orders of magnitude higher than existing durable storage technologies necessitates a reevaluation of existing software techniques. Traditionally, applications requiring data to be durable used logging with periodic write-backs to storage to ensure data consistency while maintaining acceptable performance. However, with NVRAM, now it is possible for data to reside in memory durably, eliminating the need to write back data to peripheral storage devices. Furthermore, since NVRAM is byte-addressable, durable updates can be performed at a much finer granularity than before, helping to eliminate wasted work and allowing small updates to have acceptable performance.

Because of these advantages, existing applications will observe improved performance due to NVRAM. In addition, more applications are expected to leverage durable data in the future. For example, we believe that many applications will start to save or memoize values across executions. Whereas in the past, the performance overhead of storing this data in durable memory made such memoizations unprofitable, we believe that NVRAM's substantial performance improvements will now make such optimizations profitable.

To enable applications to begin to take advantage of NVRAM, many frameworks for NVRAM have been proposed, including frameworks for C/C++ [4, 12–15, 19, 30, 37] and Java [4, 39] applications. Unfortunately, existing frameworks closely mirror the underlying hardware. This lack of abstraction hurts programmer productivity, makes it easy to write buggy code, and limits the optimizations a compiler can perform. Furthermore, this low level of abstraction does not match the expectations of managed language users.

To rectify this situation, in this paper we describe a new programming model for durable applications in NVRAM which is intuitive, not prone to user bugs, and is specified loosely enough to allow language implementers to perform many optimizations while still adhering to the model. Our model is specifically geared towards managed languages and defined at a level of abstraction users of such languages have come to expect. Our model consists of four requirements that implementations must meet. The requirements ensure that data cannot be unexpectedly volatile and allow for a user to clearly reason about what data resides in NVRAM at a given execution point.

In addition to proposing this model, we also briefly describe how Java can be extended to support our proposed model. Finally, we present some initial results on the performance overheads of creating durable applications in NVRAM and describe what future work we intend to complete.

2 RELATED WORK

2.1 Ordering of Durable Stores

While NVRAM moves non-volatile memory a level closer to the processor, many levels of volatile cache still exist between the processor and NVRAM. This means that care must be taken to ensure a store from the processor becomes persistent, or, in other words, that the new value is propagated to NVRAM and not hidden by the cache hierarchy.

Without special instructions, the order in which stores are made persistent depends on the order in which they are evicted from the cache hierarchy. Instead, *persists* must be performed to ensure a store reaches NVRAM. Recently, x86-64 processors have added an new persist instruction [1] to support writing back a cacheline to non-volatile memory without flushing the cacheline. Multiple cacheline writebacks are allowed to be internally reordered by the processor unless fences are placed in the code.

Similar to how a processor's consistency model dictates when stores and loads become visible to other threads, *persistency models* have been proposed [11, 20, 22, 26, 31] to dictate how loads to and stores from non-volatile memory can be reordered. These persistency models are enforced by placing fences and persists within the application to ensure a specific ordering between operations. Different persistency models allow different amounts of reordering, with more relaxed models potentially having better performance at the cost of potentially creating very unintuitive data states in the non-volatile memory. The persistency model we propose later in the paper is derived from these existing proposals.

2.2 Existing Frameworks for NVRAM

Currently, the Storage Networking Industry Association (SNIA) has been working to standardize the interactions with NVRAM. They have created a low-level programming model [3] meant to be followed by device driver programmers and low-level library designers. In addition, an open source project led by Intel has been created to provide application developers a higher-level toolset which is compliant with their device-level model. This project has resulted in the development of the Persistent Memory Development Kit (PMDK) [4], a collection of libraries in C/C++ and Java which a developer can use to build durable applications on top of NVRAM.

PMDK requires that programmers explicitly label all durable data in their code with pragmas. As an alternative, PMDK also contains a few library data structures, such as a durable primitive array and map, with the necessary persistent pragmas already built into the library.

```
template <class E>
class DurableList{
    durable E *element;
    durable DurableList *next;

    DurableList append(durable E *element){
        durable DurableList *head =
            durable_new DurableList();
        head->element = element;
        head->next = this;
        return head;
    }
}
```

Figure 1: Example with simplified PMDK pragmas

For persistently storing durable data, PMDK requires the user to either explicitly persist stores or use demarcated failure-atomic regions. Failure-atomic regions enables many stores to persistent memory to have the appearance of being persisted atomically. Recently, PMDK has also introduced C++ templates that allow some operations to be persistent without explicit user markings.

Figure 1 shows how to append to a durable list of type E using a simplified version of the PMDK template pragmas. As shown by the figure, the user is expected to mark all pointers to durable objects with the `durable` keyword. In addition, durable objects must be explicitly allocated in non-volatile memory with `durable_new`.

In addition to the industrial efforts, academia has also proposed many frameworks for NVRAM [12–15, 19, 30, 37, 39]. The level of support provided by these frameworks varies. At most, they provide a similar level of abstraction as PMDK, with the user having to specify all durable objects and also providing some minimal failure-atomic region support. We discuss the limitations of these existing frameworks in Section 3.

2.3 Persistent Programming Languages

Many persistent programming languages [6, 10, 17, 21, 35, 36] and implementations [7, 24, 28] were proposed before the introduction of byte-addressable non-volatile memory. Many of these languages focus on attaining the *orthogonal persistence* defined by Atkinson and Morrison [8], where the persistency of an application is orthogonal to its design.

Previous persistent programming languages are designed for a two-level storage model with orders of magnitude differences in performance between volatile and non-volatile storage. In these systems, the volatile memory is used as a cache for the non-volatile storage and much effort is devoted towards having an effective and scalable object storage model [32], implementing efficient barriers [18, 25], and optimizations such as pointer swizzling [29]. Futhermore, due to the two-level storage, persistence is attained via checkpoints whose frequency largely determines the application's performance.

The model we propose in this paper is different from previous works. Our model does not seek to attain complete orthogonal persistence and it is designed for present-day NVRAM systems. Since NVRAM provides persistency at the main memory level, instead of providing checkpoints at intervals, our model allows for continual updates to the persistent state. In addition, because our model does not use whole-application checkpoints, it expects the user to be aware of which objects are needed at recovery time to recreate the program state. However, like previous proposals, our model also emphasizes having a simple persistent programming model from the user's perspective and using reachability to limit the programmer's burden.

2.4 Current Java Persistency Techniques

Presently, the two most popular ways to durably store objects within Java is by either using the Java Persistence API (JPA) or extending Java's `Serializable` interface [16]. JPA is an API which allows applications to transparently interface with databases from multiple providers while extending the `Serializable` interface allows an application designer to directly write objects to durable storage. These existing techniques are designed for when there is a separation between the volatile main memory and non-volatile storage. New frameworks need to be designed for Java to fully leverage the capabilities of NVRAM.

3 LIMITATIONS OF EXISTING NVRAM FRAMEWORKS

Existing frameworks with support for programming NVRAM ask programmers to make many concessions. To use them correctly, a programmer must correctly mark all memory which should be durable and ensure that data is persisted properly either through explicit failure atomic regions or persists. This is an error-prone process requiring many markings in code and prohibiting the use of preexisting libraries. As highlighted by Ren, et al. [34], programmers have many difficulties correctly adapting code to be compliant with existing NVRAM frameworks.

Such existing frameworks are incongruent with the current trend towards managed languages. Managed languages, such as Java, try to lower the programmer burden and increase both safety and productivity. Java tries to provide a user with a simple programming model. For instance, for multi-threaded programs to execute correctly in the presence of potential data races, Java provides the `synchronized` keyword and requires only that the user add the `synchronized` keyword as necessary to adhere to its Data-Race-Free (DRF) memory model [27]. Java does not force the programmer to reason about the consistency model and synchronization primitives available on the underlying platform hardware, nor does it require the programmer to alter the code to run correctly in different environments. Unlike Java synchronization, existing NVRAM frameworks are both tied to the underlying hardware and its features closely match the current hardware primitives.

Another tenet of managed languages such as Java is to ensure safe execution. Java performs many runtime checks to detect incorrect programs early before lasting damage is done. For instance, Java automatically checks array accesses to ensure the element being accessed remains within bounds and triggers an exception as soon as an out of bounds access occurs, preventing unintentionally buggy programs or malicious entities from continuing to execute and potentially leaking or corrupting memory. Contrary to Java, existing NVRAM frameworks present many opportunities for unchecked or silent errors to occur, such as if non-volatile memory points to volatile memory or if a consistent program state is not persisted before a crash occurs.

Managed languages typically rely on a large central set of libraries and utilities included by default with their distributions. Programmers appreciate that via this built-in functionality there is a de facto standard application programmer interface (API) for many data structures and template for accomplishing most tasks. Unfortunately, since existing NVRAM frameworks require each durable object to be marked, existing built-in libraries cannot be used, as they will not have the proper durable markings and persists in place. In other words, current NVRAM frameworks require either the use of third-party libraries or for the user to reimplement their needed support in a durable manner. This creates opportunities to introduce many bugs into the program and requires existing applications to undergo large rewrites to be converted into durable applications.

Java and managed languages in general try to provide the programmer with simple intuitive models which are easy for the user to adhere to. However, while the models provided may be high-level, this does not mean that users do not expect competitive performance. Indeed, users expect Java code to execute efficiently and have minimal overheads. To accomplish this, most Java/JVM implementations employ Just-In-Time (JIT) compilers with speculative optimizations to attain maximal performance. JIT compilation allows for the generated code to be optimized for the common, or "hot," paths seen during execution. Furthermore, since the models Java provides are high-level, the compiler has much freedom to perform optimizations which may benefit the current execution.

Unfortunately, low-level frameworks, such as what exists for NVRAM currently, have limited optimization potential. This is because they are overspecified – by the framework features being closely tied to existing hardware, the high-level intentions of the programmer are lost, making it hard to for a compiler to be effective. For instance, in many frameworks the user must manually perform persist operations and design the logging necessary for failure-atomic regions. This ties the application to a specific implementation of failure-atomic region support. Furthermore, if a user manually emits persist operations, they may be unnecessary or in suboptimal places. Unfortunately, the compiler will struggle to optimize around and remove them, as explicit persist operations can have barriers which limit the compiler's ability to perform optimizations.

Overall, these existing frameworks impose many restrictions on application programmers which will limit their integration into managed languages. Clearly a new NVRAM programming model is needed to match the expectations of managed language programmers.

4 NEW MODEL

In the previous section we highlighted the main deficiencies of existing NVRAM frameworks; namely, that many of their features are incongruent with the philosophy of Java and other managed languages. In this section, we now provide a high-level specification of what we believe a managed language NVRAM programming model should entail.

4.1 Model Goals

An ideal model for programming NVRAM should be very intuitive for a programmer to use, not overspecified, and should be decoupled from the underlying hardware. This allows for the model to remain unchanged as hardware improves, enables the compiler to make aggressive optimizations, and minimizes the chances for the programmer to write incorrect durable programs. Below we highlight the main goals our model should meet.

Goal 1. *As few objects as possible should require durable markings.*

Current models require programmers to mark many objects as durable. This is because they want to ensure only objects which must necessarily be durable incur the performance overheads of residing in non-volatile memory. However, this is very error-prone and requires the programmer to mark many objects. Contrary to this, we believe the number of durable markings should be minimal; a user should only have to mark objects immediately visible during the crash recovery process. We believe that the runtime should then automatically make all objects reachable from these few objects durable.

Goal 2. *Libraries and other pre-existing codes should not need to be changed to work correctly in a durable program.*

As described in Section 3, existing NVRAM frameworks cannot be used with current unmodified standard libraries. We believe this is unacceptable – users should not be forced to rewrite large swaths of code and use unfamiliar libraries to create durable applications.

Goal 3. *The user should not need to explicitly persist durable objects.*

Many current NVRAM frameworks require the user to explicitly persist objects to ensure a value reaches NVRAM. This limits the amount of optimizations the compiler can perform and potentially enables the user to either add an excessive or insufficient number of persist operations. We believe that a framework should automatically persist durable objects as necessary without user involvement.

Goal 4. *A clear and simple persistency model should be provided.*

As described in Section 2.1, the order of operations to NVRAM may not be in program order unless measures are taken, due to caches in between the processor and NVRAM. This can result in program state at recovery time that does not correspond to a sequential execution of the application. A persistency model must be established for the framework that is intuitive to the user and simplifies recovery.

Goal 5. *Failure-atomic region support should be provided and need only minimal markings.*

In many cases, it is necessary for a region of code to appear to execute atomically in case of failure, with either all or none of the operations in the region being persisted. We believe support for failure-atomic regions must be provided, and that it should be intuitive for programmers to use. Namely, the user should not need to differentiate between durable and volatile objects within the region and the mechanisms for achieving this atomicity should be transparent.

4.2 Establishing New NVRAM Programming Model

With the above goals in mind, we now establish a new NVRAM programming model for managed languages. Our model consists of four requirements we believe a managed language NVRAM framework should uphold. We believe that it is the *runtime's* obligation to ensure they hold true; in other words, the runtime, not the user, must perform actions to meet these requirements. The requirements we create fall into two categories: determining which objects must be placed in non-volatile memory and ensuring the order in which stores are persisted is intuitive to programmers.

4.2.1 Placing Objects in Non-Volatile Memory.

NVRAM Model Requirement 1. *All objects reachable from the durable root set must be recoverable and in non-volatile memory.*

We define the *durable root set* as the set of pointers which are named entries into durable structures. At recovery time, the programmer can directly access these roots by name. Since these roots are visible across executions, by necessity they must named and marked; otherwise, they cannot be recovered if a crash were to occur.

This requirement helps to meet Goals 1 and 2. This requirement helps to meet Goal 1 because it only requires that the durable root set have markings; since all of objects reachable from this set must be stored in non-volatile memory by this requirement, it is unnecessary to mark them. Note that all objects which should be durable must be reachable from a durable root; otherwise, it would be impossible to access them across executions as they are unnamed.

This requirement also helps meet Goal 2, as this requirement implies that if a library data structure is reachable from a durable root, then it will automatically be made durable. This prevents the libraries from having to be modified in any way. Specifically, built-in classes' fields do not need durable markings as is necessary in existing NVRAM frameworks.

To meet this requirement, the runtime may need to move objects to non-volatile memory when it detects they are reachable from a durable root. Note that managed languages already move objects throughout execution while performing garbage collection. How the runtime chooses to adhere to this requirement should be implementation specific.

4.2.2 Controlling Persistent Atomicity Granularity.

NVRAM Model Requirement 2. *Support for failure-atomic regions must be provided. All stores to durable objects within a failure-atomic region should appear to have been performed atomically and persistently at the end of the region.*

This requirement is intended to satisfy Goal 5. Namely, this requirement ensures support for atomic regions is provided, users do not have to explicitly mark objects within atomic regions, and that failure-atomic regions' behavior is as expected.

While this requirement ensures that users have the support for failure-atomic regions of arbitrary size they expect, it also does not place unnecessary limitations on the language runtime and compiler. The runtime is free to perform any logging strategy and the compiler is free to reorder operations to both volatile and non-volatile memories as long as the model requirements are met.

NVRAM Model Requirement 3. *Outside of explicit failure-atomic regions, each store to memory reachable from a durable root should be persistently completed before a new store to non-volatile memory can proceed.*

This requirement helps to meet Goals 3 and 4. First, this requirement ensures stores to durable objects must be persistently performed without explicit user instructions. Second, for a single-thread, this requirement enforces a specific ordering of stores to NVRAM. This allows the user to clearly reason about what values will be persisted at a given point in the execution.

To meet this requirement the runtime is responsible for inserting persist operations and fences as necessary. Like NVRAM Model Requirement 1, how the runtime chooses to achieve this should be implementation specific.

NVRAM Model Requirement 4. *All stores to durable objects within a failure-atomic region should become instantaneously visible to other threads at the end of the region.*

The requirement helps to meet Goal 4. In the absence of such a requirement it is unclear what version of the value a thread will read from a shared durable object as the object is being modified by another thread within a failure-atomic region. This requirement helps to maintain failure-atomic region isolation and ensure situations do not arise where causality is violated.

To meet this requirement the runtime must monitor accesses to shared objects currently being manipulated within any failure-atomic regions and direct a given thread to the proper version of the shared object. Once again, how the runtime chooses to achieve this should be implementation specific.

5 APPLYING OUR MODEL TO JAVA

Given the NVRAM programming model requirements proposed previously, in this section we briefly describe how Java can be extended to support these requirements. Note that here we only describe the *beginnings* of extending Java for writing durable applications in NVRAM. A more rigorous description of a NVRAM programming model extension for Java is left as future work.

5.1 Marking Durable Roots

We first describe our approach to labeling durable roots in Java. Instead of adding additional keywords to Java, we propose using annotations [16]. Durable roots are to be labeled with the `@durable_root` annotation and are only allowed to be linked to *static fields*. The rationale for limiting durable roots to static fields is that durable roots must be recoverable after a crash. Like static fields, durable roots require a unique name in the environment for the recovery process. Having a normal object field as a durable root is problematic, as the field would be tied to a specific instance of the object, whereas the static field will only have one instance.

5.2 Default Persistency Support

To meet NVRAM Model Requirement 3, stores to durable objects outside of explicit failure-atomic regions should be persisted in program order. We apply this requirement to Java by dictating that persistent stores to fields of durable objects complete in program order. Note that both stores to non-durable objects and local primitive variables are still allowed to be reordered to the same degree as previously specified in the Java Memory Model [27]. This is because both non-durable objects and primitive variables will be unreachable from a durable root and hence be unrecoverable. While difficult to differentiate between durable and non-durable objects statically, speculative optimizations can be used to ensure non-durable objects can be reordered to the fullest extent of the Java Memory Model.

5.3 Affected JVM Bytecodes

To meet the requirements imposed by the changes to Java described above, the semantics of several JVM bytecodes must be changed. Below we describe the key changes to the bytecodes used when storing to object fields and arrays.

PUTFIELD: Normally this instruction stores value V into field F of object O. Now, this bytecode must now first check to see if O is a durable object. If O is not durable, then the operation proceeds as before. However, If O is durable, then storing V must be persisted in program order relative to other stores to durable objects. In addition, if V is a reference, then the instruction must also ensure V points to a durable object. If the object ($O_{volatile}$) V points to is not currently durable, then $O_{volatile}$ as well as everything reachable from $O_{volatile}$ must be moved to durable storage.

PUTSTATIC: Normally this instruction stores value V into field F of static object O. If the field F is not a labeled durable root, then `putstatic`'s semantics are unchanged. However, if field F is annotated with the `@durable_root` marking, then the value pointed to by V should be made durable if necessary, as if O itself were to be a durable object.

(A,B,C,D,F,I,L,S)ASTORE: Normally this family of instructions stores a value V into array A of type T at index I, with the specific instruction used dependent on type T. The instructions' new semantics are similar to those defined for `putfield`. Note that only `aastore` must check if the value pointed to by V is durable or not, as the other primitive types are copied by value.

6 EVALUATION

To analyze the potential performance impact of our NVRAM programming model in Java, we evaluate the DaCapo Benchmark Suite [9] with three configurations.

The initial configuration, *Baseline* (B), is the unmodified JVM. Configuration *WChecks* (C), is the support we proposed in Section 5. This configuration moves objects as necessary to NVRAM to meet NVM Model Requirement 1 and orders persistent stores according to NVM Model Requirement 3. In addition, this configuration performs the new necessary checks on the affected bytecodes to determine what the behavior of the bytecode should be. Configuration *AllDurable* (A), is like configuration C, but assumes all objects should be treated as durable objects. While our NVRAM programming model does not forbid doing this, treating all objects as durable objects can have many overheads, including unnecessarily following our persistency model and persistently storing much data that cannot be recovered.

We modify the Maxine Java Virtual Machine [38] to implement these configurations. Maxine is a research JVM designed to enable fast prototyping of new features while still achieving competitive performance. We are currently using Maxine 2.0 and have modified its first-tier compiler (T1X) to implement the changes proposed in Section 5. We limit the use of its second tier compiler (C1X) to code regions unable to be compiled by T1X.

We run each of these configurations on the DaCapo Benchmark Suite. While this benchmark suite does not contain durable applications, it is sufficient to test the overheads of our extensions to Java. Due to the nature of the configurations, configuration C models the

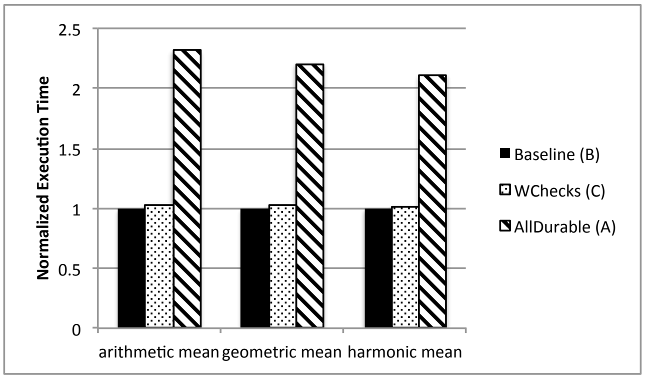

Figure 2: Average Normalized Execution Time for the DaCapo Benchmarks

overheads of performing the checks for durable objects required for the modified bytecodes while configuration *A* treats all objects as if they must be durable. Hence, configuration *A* acts as a worst case when all data within an application should be made durable while configuration *C* shows the overhead of the extra checks of the affected bytecodes on all objects.

We run our evaluation on a machine with an 8-core Intel Skylake i7-7820X CPU and 32GB of DDR4 DRAM. While we currently do not use NVRAM, we accurately model the main overheads of durable applications by performing the necessary persist operations and using fences to enforce our persistency model. For writing cachelines back to memory we use Intel's `CLWB` instruction [1] and we use the `SFENCE` to implement persistent fences.

Figure 2 shows the arithmetic, geometric, and harmonic means of our DaCapo performance results normalized to *B*. On average, configuration *C* has negligible overheads while configuration *A* has 131%, 120%, and 111% overheads for the arithmetic, geometric, and harmonic means, respectively. A primary reason *C* has minimal overheads is that overheads inherent in the T1X compiler are large enough to make the overheads of durable checks minor. Likewise, the overheads of *A* will likely be larger in a highly optimized system unless measures are taken. However, we believe that additional compiler optimizations will be able to limit the overheads of our persistency model. For instance, it is possible to make speculative optimizations based on past execution and to recompile if these assumptions are invalidated.

Overall, we believe these are promising initial results. We plan to create and test true durable applications once we have fully defined and implemented our NVRAM extension to Java.

7 FUTURE WORK

As mentioned in Section 5, the changes we propose in this paper are only the beginnings of extending Java to enable writing durable applications in NVRAM. In the future, we plan to fully define a NVRAM programming model in Java which adheres to all requirements described in Section 4.2. This includes adding support for failure-atomic regions, introspection of durable objects, naming durable roots, an object recovery API, and a clear multi-threading persistency model.

In addition to the extensions to Java, we also plan to fully describe how the JVM should be modified to accommodate our Java extensions. This includes modifying additional bytecodes, possibly adding new bytecodes, and adding new internal metadata structures. We also must define new non-volatile heap regions as well as the new expectations of the garbage collector. By fully defining the support necessary at the JVM level, it will be possible for other JVM-based languages such as Clojure, Kotlin, and Scala to also add support for writing durable applications in NVRAM.

Finally, once we have a fully defined NVRAM programming model for both Java and the JVM, we plan to implement a full version of our model within the Maxine VM. We plan to adapt both Maxine's first tier (T1X) and second tier (C1X or Graal) compilers to match our new model. Also, we plan to introduce optimizations which can use profiling information gathered from the first tier to create more efficient code and reduce the overheads of writing durable applications.

8 CONCLUSION

In this paper we described the limitations of current NVRAM programming models. In addition, we proposed a new high-level NVRAM programming model for managed languages.

After proposing a new NVRAM programming model, we briefly described how Java could be extended to support this model. Finally, we presented some initial results on the performance overheads of creating durable applications in NVRAM and described our future work.

REFERENCES

[1] 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
[2] Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html
[3] NVM Programming Model v1.2. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf
[4] Persistent Memory Development Kit. http://pmem.io/pmdk/
[5] H. Akinaga and H. Shima. 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. *Proc. IEEE* 98, 12 (Dec 2010), 2237–2251. https://doi.org/10.1109/JPROC.2010.2070830
[6] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. 1982. PS-algol: An Algol with a Persistent Heap. *SIGPLAN Not.* 17, 7 (July 1982), 24–31. https://doi.org/10.1145/988376.988378
[7] Malcolm Atkinson and Mick Jordan. 2000. *A Review of the Rationale and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform.* Technical Report. Mountain View, CA, USA.
[8] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally Persistent Object Systems. *The VLDB Journal* 4, 3 (July 1995), 319–402. http://dl.acm.org/citation.cfm?id=615224.615226
[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06).* ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488
[10] Luc Bläser. 2007. Persistent Oberon: A Programming Language with Integrated Persistence. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–85.
[11] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016).* ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704
[12] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).* ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[13] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[14] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[15] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. 2016. 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 (HPDC '16)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/2907294.2907303

[16] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[17] Antony L. Hosking and Jiawan Chen. 1999. PM3: An Orthogonal Persistent Systems Programming Language - Design, Implementation, Performance. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 587–598. http://dl.acm.org/citation.cfm?id=645925.671503

[18] Antony L. Hosking, Nathaniel Nystrom, Quintin I. Cutts, and Kumar Brahnmath. 1999. Optimizing the Read and Write Barriers for Orthogonal Persistence. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 149–159. http://dl.acm.org/citation.cfm?id=648123.747258

[19] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[20] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

[21] Mick Jordan and Malcolm Atkinson. 2000. *Orthogonal Persistence for the Java[Tm] Platform: Specification and Rationale*. Technical Report. Mountain View, CA, USA.

[22] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[23] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan 2010), 143–143. https://doi.org/10.1109/MM.2010.24

[24] Brian Lewis, Bernd Mathiske, and Neal M. Gafter. 2001. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. In *Revised Papers from the 9th International Workshop on Persistent Object Systems (POS-9)*. Springer-Verlag, London, UK, UK, 18–33. http://dl.acm.org/citation.cfm?id=648124.747405

[25] Brian T. Lewis and Bernd Mathiske. 1999. *Efficient Barriers for Persistent Object Caching in a High-Performance JavaTM Virtual Machine*. Technical Report. Mountain View, CA, USA.

[26] Y. Lu, J. Shu, L. Sun, and O. Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[27] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

[28] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. 2001. Implementing Orthogonally Persistent Java. In *Revised Papers from the 9th International Workshop on Persistent Object Systems (POS-9)*. Springer-Verlag, London, UK, UK, 247–261. http://dl.acm.org/citation.cfm?id=648124.747395

[29] J. E. B. Moss. 1992. Working with persistent objects: to swizzle or not to swizzle. *IEEE Transactions on Software Engineering* 18, 8 (Aug 1992), 657–673. https://doi.org/10.1109/32.153378

[30] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. 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 '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[31] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[32] Tony Printezis, Malcolm Atkinson, Laurent Daynés, Susan Spence, and Pete Bailey. 1997. *The Design of a new Persistent Object Store for PJama*. Technical Report. Mountain View, CA, USA.

[33] 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. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (July 2008), 465–479. https://doi.org/10.1147/rd.524.0465

[34] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. ACM, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3124680.3124729

[35] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. 1993. The Design of the E Programming Language. *ACM Trans. Program. Lang. Syst.* 15, 3 (July 1993), 494–534. https://doi.org/10.1145/169683.174157

[36] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. 1993. Texas: An Efficient, Portable Persistent Store. In *Persistent Object Systems*, Antonio Albano and Ron Morrison (Eds.). Springer London, London, 11–33.

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

[38] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

[39] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. 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 (ASPLOS '18)*. ACM, New York, NY, USA, 70–83. https://doi.org/10.1145/3173162.3173201