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 amount of optimizations the 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 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 implementors 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.

1 INTRODUCTION
In recent years, technological advances have been made towards having byte-addressable non-volatile memory. Whereas traditionally durably storing data require 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) [16, 21] and Resistive RAM (ReRAM) [5] are being rapidly developed that offer non-volatile memory with byte-level access granularity. 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 durable techniques. Traditionally, applications requiring data to be durable use 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.

Because of these advantages, NVRAM promises existing durable applications to have much better performance. In addition, more applications are expected to leverage durable data in future. For example, we believe the many application will start to save or memoize values across executions. Whereas in the past the performance overhead of storing this data in durable memory negated the benefits of storing this information, 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, 8–11, 13, 19, 23] and Java [4, 25] 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 amount of optimizations the 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 implementors 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 of the model must obey. The requirements ensure that data isn’t unexpectedly volatile and allows 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, there still exists levels of volatile cache between the processor and NVRAM. This means that care must be taken to ensure that a store from the processor becomes persistent, or, in other words, that the new store 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 [7, 14, 15, 17, 20] 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 stores.

Different persistency models allow different amounts of reordering, with more relaxed models creating potentially very unintuitive data states in the non-volatile memory with the tradeoffs of potentially having better performance. 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 lead 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 a durable application 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.

For persistently storing durable data, PMDK allows for the user to explicitly persist stores as well as use demarcated failure-atomic regions. Failure-atomic regions allow for many stores to persistent memory to become persistent atomically. Recently, PMDK has also introduced C++ templates to allow some persist operations without explicit user markings.

Figure 1 shows how to append to a durable list of type E information using simplified PMDK template pragmas.

```
1 template <class E>
2 class DurableList{
3   durable E *element;
4   durable DurableList *next;
5   
6   DurableList append(E *element){
7     durable DurableList *head =
8       durable_new DurableList();
9     head->element = element;
10     head->next = this;
11     return head;
12 }
```

Figure 1: Example with simplified PMDK 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 [8–11, 13, 19, 23, 25]. 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 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 [12]. JPA is intended to be an API to allow applications to 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 in [22], 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 programmer 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 its Data-Race-Free (DRF) memory model [18]. 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 tenant 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. 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 a common accepted route to accomplishing most tasks. Unfortunately, since existing NVRAM frameworks require each durable object to be marked, existing unmodified 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 the use of third-party libraries or for the user to reimplement the 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 to 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 high 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 model 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 them, as explicit persist operations can have barriers which limit the compiler’s ability to reorder code.

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 details at a high level 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 then that the runtime should then automatically make all object reachable these few objects durable as well.

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 that this is unacceptable – users should not be forced rewrite large swaths of code and use unfamiliar libraries to create a durable application.

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 too many or not enough 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, due to caches in between the processor and NVRAM, unless measures are taken, the order of stores to NVRAM may not be in program order. This can result in program state at recovery time which does not correspond to a sequential execution of the application. A persistency model must be established for the framework which is intuitive to the user and simplifies recovery.

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

In some cases it is necessary for a user to indicate that a region of code must 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. 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 region’s 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 above requirement is 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 an 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 value a thread will read from a shared durable object as the object is being modified by another thread. This requirement helps to 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 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 [12]. Durable roots are to be labeled with the @durable_root annotation and are only allowed to 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 as before. This is because both non-durable objects and primitive variables will be unreachable from a durable root.

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 \( O_{\text{volatile}} \) \( V \) points to is not currently durable, then \( O_{\text{volatile}} \) as well as everything reachable from \( O_{\text{volatile}} \) 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 new semantics are similar to those defined for putfield. However, if field \( F \) is annotated with the \texttt{@durable_root} marking, then 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)STORE**: 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 astore 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 [6] 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 model requirement 1 and orders persistent stores according to 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 which cannot be recovered.

We modify the Maxine Java Virtual Machine [24] 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 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 worse 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.

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.

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 adhering to all requirements described...
Defining a High-Level NVRAM Programming Model

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 multithreading 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 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


