Received: 11 February 2019 | Revised: 13 November 2019 | Accepted: 16 November 2019

DOI: 10.1002/spe.2781

RESEARCH ARTICLE

WILEY

Orthogonal persistence in nonvolatile memory architectures: A persistent heap design and its implementation for a Java Virtual Machine

Taciano D. Perez[1] | **Marcelo V. Neves**[2] | **Diego Medaglia**[3] | **Pedro H. G. Monteiro**[3] | **César A. F. De Rose**[2]

[1]ASML, Eindhoven, The Netherlands

[2]Polytechnic School, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil

[3]HP Research & Development, Porto Alegre, Brazil

Correspondence
Marcelo V. Neves, Polytechnic School, Pontifical Catholic University of Rio Grande do Sul, 906197-900 Porto Alegre-RS, Brazil.
Email: marcelo.neves@pucrs.br

Summary

Current computer systems separate main memory from storage, and programming languages typically reflect this distinction using different representations for data in memory and storage. However, moving data back and forth between these different layers and representations compromise both programming and execution efficiency. To remedy this, the concept of *orthogonal persistence* (OP) was proposed in the early 1980s advocating that, from a programmer's standpoint, there should be no differences in the way that short-term and long-term data are manipulated. However, at that time, the underlying implementations still had to cope with the complexity of moving data across memory and storage. Today, recent nonvolatile memory (NVM) technologies, such as resistive RAM and phase-change memory, allow main memory and storage to be collapsed into a single layer of persistent memory, opening the way for more efficient programming abstractions for handling persistence. In this work, we revisit OP concepts in the context of NVM architectures and propose a persistent heap design for languages with automatic memory management. We demonstrate how it can significantly increase programmer and execution efficiency, removing the impedance mismatch of crossing semantic boundaries. To validate and demonstrate the presented concepts, we present JaphaVM, an implementation of the proposed design based on JamVM, an open-source Java Virtual Machine. Our results show that JaphaVM, in most cases, executes the same operations between one and two orders of magnitude faster than regular database-based and file-based implementations, while requiring significantly less lines of code.

KEYWORDS

Java Virtual Machine, nonvolatile memory, orthogonal persistence, persistent memory

1 | INTRODUCTION

Computer systems traditionally separate main memory from storage. This distinction is imposed by limitations of access latency, cost, volatility, power, or capacity of the existing memory and storage technologies. Programming languages typically reflect this distinction using semantically different representations for data in memory (eg, data structures and

objects) and in storage (eg, files and databases). Moving data back and forth between these different layers and representations has implications for programmer efficiency (additional effort, complexity, maintenance challenges, and probability of defects) as well as for execution efficiency (unnecessary data movement and duplication, increased number of instruction cycles, and memory/storage usage). Data-type protection offered by programming languages is also often lost across this mapping. This problem, dubbed *impedance mismatch*, has been described in the literature as the *Vietnam of Computer Science*.[1]

Based on these observations, the concept of *orthogonal persistence* (OP) was proposed in the early 1980s.[2] It proposes that from a programmer's standpoint, there should be no differences in the way that short-term and long-term data are manipulated. In other words, persistence should be an *orthogonal property of data*, independent of data type and the way in which data is handled. Programmers should focus on the core aspects of their applications, while the runtime environment would take care of managing the longevity of data. During the 1980s and 1990s, this concept was explored in several research initiatives, including programming languages, operating systems, and object-oriented databases.[1,3] However, the underlying implementations still had to cope with the complexity of moving data across memory and storage.

Recent byte-addressable, nonvolatile memory (NVM) technologies such as phase-change RAM[4,5] and memristor[6] are expected to enable memory devices that are nonvolatile, require low energy, and have density and latency closer to dynamic RAM (DRAM). These technologies allow main memory and storage to be collapsed into a single entity: *persistent memory (PM)*. It has potential to improve file systems, databases, operating systems, and any other system relying on data storage, including programming languages, the focus of this work.

Many recent studies proposed *PM programming* interfaces to manipulate data in byte-addressable, NVM.[7] So far, interfaces were proposed for languages with explicit memory management, such as C and C++.[8-12] However, little work has been carried out to explore PM programming interfaces in languages with managed runtimes and automatic memory management, such as Java, Python, Ruby, and JavaScript. These languages offer interesting potential for implementing programming interfaces that handle persistence transparently, since the existing garbage collection (GC) mechanisms can be leveraged to determine the appropriate object lifetime. The present work helps filling this gap, by presenting the first Java Virtual Machine (JVM) specifically designed for PM, leveraging automatic memory management to enable automatic data persistence between JVM reinitializations.

The main contributions of this work are described as follows. First, we revisit OP concepts in the context of PM, and propose a design for the runtime environment of languages with automatic memory management based on an original combination of OP, PM programming, persistence by reachability, and lock-based failure-atomic transactions. We demonstrate how it can significantly increase programmer and execution efficiency, removing the impedance mismatch of crossing semantic boundaries. Second, we present JaphaVM, an implementation of the proposed design based on JamVM,[13] an open-source JVM, to validate and demonstrate the presented concepts. To the best of our knowledge, JaphaVM is the first JVM specially designed to take advantage of NVM technologies.

The rest of the paper is organized as follows. Section 2 outlines background information related to OP and associated principles and challenges. Section 3 presents the proposed design, and Section 4 shows the JaphaVM prototype implementation. Section 5 presents the evaluation results. We then review and put our contributions in the context of related research in Section 6. The conclusion and future work are presented in Section 7.

2 | ORTHOGONAL PERSISTENCE

As mentioned previously, the cost of mapping data as represented in memory to either files or databases (and vice-versa) is known as *impedance mismatch* and adds complexity to both software development and execution.[1] Orthogonally persistent object systems propose to solve this problem by supporting a uniform treatment of objects irrespective of their types, allowing values of all types to have whatever longevity is required. The benefits of OP can be summarized as follows:

- Improving programming productivity from simpler semantics.
- Avoiding ad hoc arrangements for data translation and long-term data storage.
- Providing protection mechanisms over the whole environment.
- Supporting incremental evolution.
- Automatically preserving referential integrity over the entire computational environment for the whole lifetime of an application.

Atkinson and Morrison[14] identified three principles that, if applied, would yield OP, summarized as follows.

1. *The principle of persistence independence*. The form of a program is independent of the longevity of the data it manipulates. Programs look the same whether they manipulate short-term or long-term data.
2. *The principle of data type orthogonality*. All data objects should be allowed the full range of persistence irrespective of their type. There are no special cases where objects are not allowed to be long-lived or are not allowed to be transient.
3. *The principle of persistence identification*. The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system. The mechanism for identifying persistent objects is not related to the type system.

Experience with the development of different OP systems demonstrated that *persistence by reachability*, also known as *transitive persistence*, is the appropriate mechanism for implementing OP in systems with the work of Dearle et al.[1] It determines that all objects that are reachable directly or indirectly from a set of *persistence roots* must be treated as persistent.

Experimental implementations of the OP principles were made with several languages, including Java. Examples of Java OP implementations are Grasshopper/Java,[15] PJama,[16] PEVM,[17] ANU OPJ,[18] Merpati[19], and aspect-oriented programming implementations.[20,21] All these systems removed the burden of explicitly translating data representations between memory and storage from the programmer; however, the problem was transferred to the internal system implementation, which still had to cope with moving data between these different domains.

The introduction of PM can solve the impedance mismatch problem at both interface and implementation levels by removing the duality memory versus storage, opening up the way for new solutions, such as our design presented in the next section.

3 | PROPOSED DESIGN

The use of one set of abstractions for handling data in memory and another for storing the same data is not a fundamental necessity of programming languages, but an accidental result of the current technology, which separates fast, small, volatile main memory from slow, abundant, nonvolatile secondary storage. If these two different layers are collapsed into PM at the hardware level, then OP implementations in software can provide simpler programming abstractions, resulting in less code complexity and lower programmer effort.

This section introduces a design of orthogonally persistent heaps for data persistence in languages with automatic memory management, assuming the availability of PM hardware. Our design can significantly increase programmer and execution efficiency, as in-memory data structures are transparently persistent, without the need for programmatic storage handling, and there is no longer a need for crossing semantic boundaries anymore.

We defined three main requirements for the proposed design. First, it should allow for simplified development and maintenance of persistence code, by adhering to the OP principles. Second, it should keep backward source and bytecode compatibility with existing code whenever possible. Finally, the proposed design should allow generalization, so it could be applied to object-oriented languages with automatic memory management (ie, not limited to Java).

We also defined some system assumptions for the target hardware platform we are aiming in: NVM devices are accessed directly using CPU load and store operations, volatile processor caches are still present, and NVM load/store latencies are similar to DRAM (an assumption also made by previous research[10,11]).

From an operating system perspective, we assume it uses a PM-aware file system, which is designed specifically for PM.[22] A PM-aware file system provides direct access to PM, removing the block I/O layer, bypassing the page cache, and eliminating other steps rendered unnecessary by PM, as data can be directly addressed by the processor at byte granularity. Examples of such file systems are BPFS,[23] PRAMFS,[24] PMFS,[25] Ext4/DAX,[26] SCMFS,[27] NOVA,[28] and M1FS.[29]

The remaining of this section describes the design of persistent heaps adherent to the OP concepts. By persistent heap, we mean all the persistent state that can be bound to a program, which may include the heap itself and other elements of the persistent state, such as threads, stacks, loaded types (classes), JVM metadata, etc. This section also lists the main challenges and open points that must be addressed to have a robust solution.

3.1 | Persistent heap adherent to OP concepts

A key component of our design is a persistent heap mapped to PM. We are considering the term *persistent heap* as referring to all state that can be made persistent, which may include not only the heap itself, but other elements, such as threads, stacks, type definitions, method code, and other data as required by the particular implementation.

To determine the longevity of objects, we advocate the use of *persistence by reachability* using as *roots of persistence* the static variables of all classes, which are persistent and the stack of all active threads. This approach leverages the same reachability criteria used for Java GC[30] and has been used in previous OP Java implementations.[15-18]

In the next sections, we list what needs to be stored in the persistent heap to support persistence by reachability of both class attributes and stack references. Although we use examples in Java to demonstrate the concepts, this approach can be generalized to other object-oriented languages.

3.1.1 | Persistence by class attribute reachability

A persistent heap in which persistence is defined by class attribute reachability contains at least the following elements:

- Type (classes, interfaces, etc) definitions
- Objects (type instances)

In this variant, only objects referenced from class attributes are considered persistent and must be spared by the Garbage Collector. When a program with a persistent heap of this variant is executed, class definitions that are already present on the persistent heap will not be loaded again, and their class attributes point to objects already present on the persistent heap.

Direct references. Figure 1 shows an example of persistence by class attribute reachability in Java. Line 3 declares the static attribute `aString`. When the class is loaded, the value of `aString` is set to the empty string "". When the method `main()` is executed for the first time, the initial value of `aString` is concatenated to the string "`AB`" (Line 6), and thus `aString` points to a new `String` object with the value "`AB`" (it points to a new object since strings are immutable objects in Java). When the program is executed for the second time, it will create a third `String` object with the value "`ABAB`." At this point, only this last `String` object is reachable from a class attribute and thus should be made persistent, while the other strings previously created are disposable. This sequence of steps is depicted in Figure 2.

```
1 public class PersistenceExample {
2
3         private static String aString = "";
4
5         public static void main(String[] args) {
6                 aString += "AB";
7                 System.out.println("aString="+aString);
8         }
9 }
```

FIGURE 1 An example of persistence by class attribute reachability

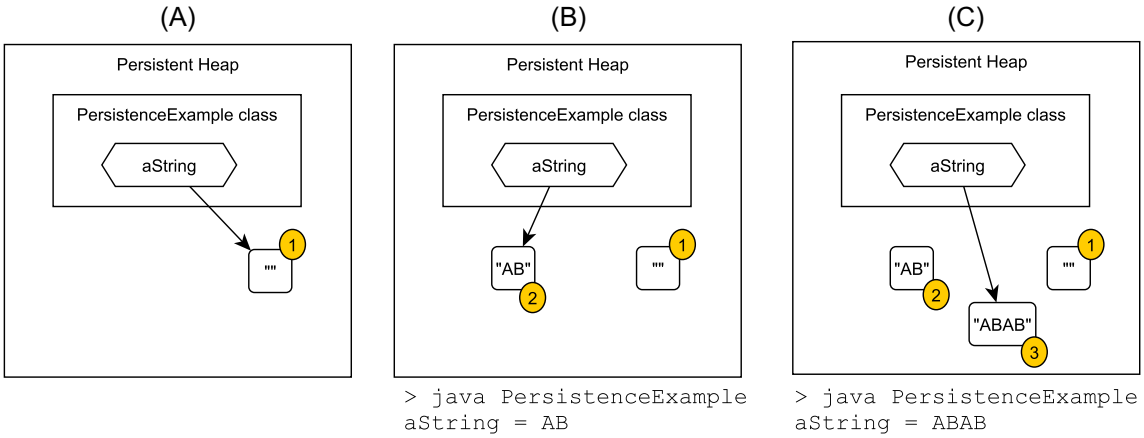


FIGURE 2 Example of persistence by class attribute. A, After persistence example class loading; B, After first execution of main(); C, After second execution of main() [Colour figure can be viewed at wileyonlinelibrary.com]

```
1  public class Proxy {
2
3          public String aString;
4
5          Proxy() {
6                  aString = "";
7          }
8  }
9
10 public class IndirectExample {
11
12         private static Proxy proxy = new Proxy();
13
14         public static void main(String[] args) {
15                 proxy.aString += "AB";
16                 System.out.println("aString="+proxy.aString);
17         }
18 }
```

FIGURE 3 An example of persistence by indirect class attribute reachability

It should be noted that this behavior deviates from the current Java language specification, since data from previous executions are preserved. This conflicts with our requirement of keeping backward source and bytecode compatibility with existing code whenever possible. Our solution to reduce the impact is offering control whether a program execution is bound to a persistent heap (thus exhibiting the behavior described here) or not (default Java semantics). The usage and management of persistent heaps and how to bind them to programs is further discussed in 3.2.

Transitive references. In the previous example, objects that were directly referenced by class attributes were considered persistent. It is also true for objects that are transitively referenced by class attributes as well. Figure 3 illustrates this case. `Proxy` is a simple class that contains one instance attribute called `aString`. Class `IndirectExample` has one class attribute of the type `Proxy` called `proxy`. Every time the method `main()` is invoked, it concatenates "`AB`" to `proxy.aString` value (Line 11). Since strings are immutable in Java, every concatenation generates a new `String` object in the heap. This sequence of steps is similar to the previous example, but now the references from a class attribute are indirect.

3.1.2 | Persistence by stack reachability

Persistence by class attribute, described in the previous section, provides a way to access persistent data created during previous program executions. By adding persistence by stack reachability, it becomes possible to create a snapshot, or checkpoint, of a given execution state and make it persistent, so it can be resumed in the future.

In this variant, the stack is also part of the persistent state, and objects that are directly or indirectly referenced by any of the active stack frames are considered persistent and are not disposable by the Garbage Collector. A persistent heap in which persistence is defined by stack reachability contains the following elements:

- Type (classes, interfaces, etc) definitions
- Objects (type instances)
- Stack
- Threads

Figure 4 shows an example of persistence by stack reachability in Java. Line 12 creates a new `stackExample` object in the heap. The stack frame of the `main()` method contains a variable `ex` referencing the `stackExample` object, so it will be considered persistent until method `main()` completes. Line 13 calls method `doConcat()`, thus creating a new stack frame. Line 4 declares string aString, initialized to value "". Lines 5-8 list a loop that will concatenate "`AB`" to `aString`, for two iterations. Every loop iteration will create a new `String` object in the heap (since strings are immutable in Java), but only the string object referenced by `aString` will be considered persistent, since it is referenced by a valid stack frame. When `doConcat()` finishes its execution, all string objects will be disposable, since no active stack frame references it. This sequence of steps is depicted in Figure 5. For example, if the execution is suspended at any of the intermediate steps (a, b, or c), the reference of `aString` in the *persistent stack* can be used to resume execution from that same step.

```
1  public class StackExample {
2
3          public void doConcat() {
4                  String aString = "";
5                  for (int i=0;i<2;i++) {
6                          aString += "AB";
7                          System.out.println(aString);
8                  }
9          }
10
11         public static void main(String[] args) {
12                 StackExample ex = new StackExample();
13                 ex.doConcat();
14         }
15 }
```

FIGURE 4 An example of persistence by stack reachability

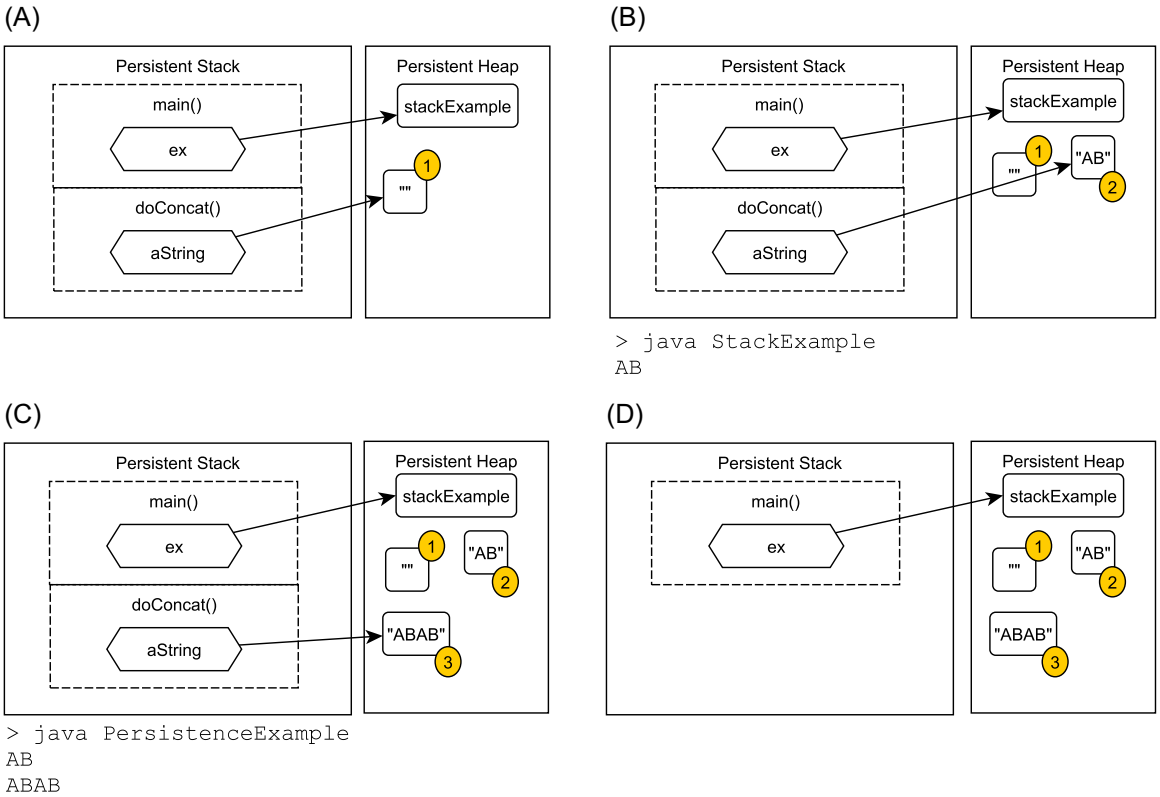


FIGURE 5 Example of persistence by stack reference. A, Execution after a string initialization (line 4); B, After first execution of loop (line 5); C, After second execution of loop (line 5); D, After method doConcat() finishes [Colour figure can be viewed at wileyonlinelibrary.com]

Persistence by stack reachability allows the complete execution state of a program to be automatically made persistent, so execution can be resumed from the same point in a future moment. If the execution is suspended or interrupted at an arbitrary moment, it can be resumed exactly from that point.

3.2 | Storage and management of persistent heaps

As mentioned, throughout this text, we use the term *persistent heap* to refer to the persistent state that can be bound to a program, which may include not only the heap itself, but all other elements of the persistent state (eg, threads, stacks, classes, and JVM metadata) as well as other data as required by the implementation.

We assume a PM-aware file system providing namespace management and user access control to chunks of PM.[7] The file system contains persistent heap files, which can be mapped to the address space of any process via *PM programming*,

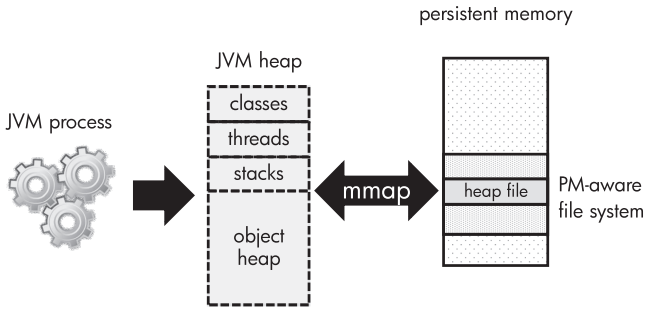


FIGURE 6 Persistent memory hosting an in-memory file system, which contains a persistent heap file

either by directly calling the `mmap()` system call or using higher-level APIs for PM.[8-10,12] Once mapped to the address space, the process can manipulate PM directly, without page caches. This scenario is depicted on Figure 6. When the runtime environment of a program is invoked (eg, JVM), the file containing the persistent heap is passed as a parameter.

In this work, we are exploring only the scenario where one process is bound to a single persistent heap at a given point in time. It is an interesting research problem for future work to consider multiple persistent heaps manipulated simultaneously by a single process, as well as multiple processes concurrently using a single persistent heap (ie, a shared persistent heap).

The persistent heap can contain only persistent data, or it can contain also a persistent execution state. Both are described in the next sections.

Data Persistence. Data persistence stores program data in a persistent heap but does not store the program execution state. After the process execution is terminated, or interrupted, all persistent data (defined by class attribute reachability, as described in Section 3.1.1) is available to be bound to a new process, that can be either the same program or a different one.

One way to instantiate data persistence is invoking a program while passing the persistent heap as an execution argument for it or for its runtime environment. Taking the program described in Figure 1 as an example, and assuming a JVM with support for OP, it could be called by the command line as `java -persistent-heap=heapfile.ph PersistenceExample`. In this example, the program would be bound to the persistent data state contained in `heapfile.ph`.

Execution persistence. Execution persistence implies that both data and the execution state of a given program are contained by the persistent heap. In this case, both persistence by class attribute reachability and by stack reachability (described respectively in Sections 3.1.1 and 3.1.2) are used. The persistent heap contains both data and program code. After the process is terminated, or interrupted, the persistent heap contains a snapshot of the execution state at a specific consistent point in time, and can be resumed from that point.

3.3 | Consistency and transactional properties

The persistent heap must be kept in a consistent state even in the presence of failures, such as abnormal program termination due to power outages, runtime faults, resource exhaustion, etc. To accomplish this, we propose the use of *failure-atomic transactions*, provided by lower-level PM programming APIs such as Mnemosyne[8] and NVML,[12] which provide failure-atomic transactions by means of transactional memory and journaling, respectively. Failure-atomic transaction implementations ensure that data is flushed from processor caches into PM with atomicity and consistency properties, so the persistent heap is not corrupted.

It is also desirable for the programmer to be able to express application-specific transactional semantics. We advocate the use of locks defined by the application programmer to identify the scope of failure atomic transactions (*lock-based failure-atomic transactions*). In the context of Java, `synchronized` methods and blocks define locks around specific objects by means of calls to `monitorenter` and `monitorexit` opcodes. Atlas[10] introduces a similar approach for the C language, and Welc et al[31] demonstrated that Java synchronization locks can be used to express transactional semantics.

3.4 | Design challenges

The previous sections described a persistent heap design for PM. This section lists the main challenges and open points that must be addressed to have a robust solution. In our solution, we have addressed some of these points, as will be

described in Section 4, and others will be target of future work.

- *Sharing data across different programs and programming languages*. It is often necessary to share persistent data across multiple distinct programs, sometimes with concurrent access. These programs can be written in the same programming language or in different programming languages.
- *Type evolution*. Each object in a heap belongs to a specific type. When a reference type is changed, objects that already exist in the heap must be bound to a new type, and its data and code must be adjusted appropriately to convey the semantics intended by the programmer. This process is known as type or class evolution.[32]
- *Persistent bugs*. A system with execution persistence may incur in persistent deterministic bugs that crash or impair subsequent executions.
- *External state*. Some objects and variables represent state that is external to the virtual machine. Notable examples are networking resources (eg, sockets), files, and elements that vary from one system to another, such as locale. There is a fundamental incompatibility arising from the fact that OP assumes that all state is persistent by default, while in a transient program all variables are reinitialized on each run.
- *Security*. When using a persistent heap, potentially sensitive data becomes vulnerable to unauthorized access and tampering. Data protection techniques such as encryption and signing are potential solutions.

4 | JAPHAVM PROTOTYPE IMPLEMENTATION

In this section, we present JaphaVM, a prototype to validate and evaluate the effectiveness of the design presented in Section 3. The current version of JaphaVM is limited to *data persistence* (described in Section 3.2). *Execution persistence* will be implemented in future versions. The full source code is publicly available at our repository.[33]

4.1 | Rationale

Many object-oriented languages can be adapted to employ the OP abstractions. We opted for Java since its use is widespread in a variety of environments, from embedded devices to high-performance computing applications, and its GC mechanism already manages the object longevity by reachability, which can be leveraged for persistence by reachability.

Some previous OP implementations in Java, such as ANU OPJ,[18] followed the approach of bytecode transformation, allowing portability across different JVMs. Other implementations, such as PJama[16] and PEVM,[17] chose to modify the JVM itself, permitting deeper modifications of the Java execution environment. We propose to use the latter approach, making modifications to the JVM, to have complete control over the data structures used to manage heap, stacks, threads, and type definitions. Since the mentioned OP-oriented JVMs are no longer available, only current JVMs can be used as implementation baseline. Our choice was JamVM,[13] due to its simple and straightforward implementation.

4.2 | JamVM and GNU classpath

JamVM runs on top of a wide set of Unix-like Operating Systems, including Linux, FreeBSD, OpenBSD, Solaris, OpenSolaris, Darwin, Android, and iOS. It runs on x86, x86-64, ARM, Sparc, MIPS, and PowerPC architectures. It was the first JVM used in the Android OS, before it was replaced by the Dalvik JVM. JamVM version 1.5.4 was used as baseline for our prototype, which was developed on Linux/x86.

JamVM requires an implementation of the Java class libraries. In this work, the GNU Classpath[34] V0.92 open-source implementation was used. Some modifications to GNU Classpath were made to address design issues, as will be described in Section 4.4.

4.3 | NVML

The baseline JamVM version uses an anonymous, nonfile-backed memory-mapped area via the `mmap` function to allocate space for the JVM heap. We attempted a first implementation of the persistent heap simply modifying the `mmap` flags to use a shared memory mapping backed by a file on top of a PM-aware file system. Although this implementation indeed worked to provide heap persistence, it did not provide fault tolerance.

Our current persistent heap implementation uses the NVML pmemobj library[12] for manipulating persistent heap data to ensure fault-tolerance, as NVML provides journaled transactions via an undo log in PM. In other words, before modifying any data in-place, NVML first adds an entry with the original, unmodified values to a log in a separate PM address space,

then modifies the value in place, and finally marks the log entry as complete; in case the program is interrupted (eg, by a power failure), the next execution will search the log for entries not marked as complete and use them to restore the original data, thus ensuring fault tolerance. To take advantage of this NVML feature, JaphaVM stores its persistent heap in an NVML *memory pool*, mapped to a file in PM. NVML has been chosen for currently being one of the most active and widely adopted PM programming APIs and providing a suitable transactional model. On advantage of NVML is not requiring memory hardware with special transactional capabilities.

4.4 | Summary of JVM modifications

The next sections present the changes made to the baseline JamVM code. Most of them were made to keep heap data and metadata in the NVML memory pool and manipulated in a way that ensures fault tolerance. Some changes were also made to the GNU Classpath library to address the handling of external state (such as standard console input/output), as further explained.

Persistent heap. Figure 6 illustrates the different data structures that a JVM stores in memory: class definition, thread data, including a stack for each thread, and the heap where object instances and their metadata are stored. As mentioned in Section 4.3, we replaced the anonymous memory mapping originally used by the JamVM by an NVML memory pool, whose contents include dynamically and statically allocated objects, Java type definitions, executable code, and additional metadata, such as heap base address, maximum heap size and free heap space, pointers to free heap regions, and GC metadata.

To allow the invocation of the JamVM with a persistent heap, we have added a new invocation argument `-persistentheap:<heapname>` that alternates between a persistent mode and nonpersistent mode.

Type definitions and compiled methods. We need to ensure that all data structures that are referenced by heap objects, such as Java type definitions and compiled methods, are made persistent and can be retrieved in future executions. However, many of these structures were stored outside of the heap allocation space in the baseline JamVM implementation. While implementing JaphaVM, we have modified its memory allocation routines to store these data structures inside the same NVML PM pool where we store heap objects.

Internal hash tables. JamVM uses a set of internal hash tables to keep track of loaded symbols, classes, and other metadata. To locate data in the PM pool across JVM executions, these tables are also stored by JaphaVM in the PM pool itself.

Failure-atomic transactions. The JVM instructions are known as *opcodes*. Some of these opcodes modify the internal VM state (eg, push/pop stack data, create new object in the heap, etc), and must never leave the heap in an inconsistent state. To meet the consistency requirements described in Section 3.3, we have modified all Java opcodes that change persistent data to happen in the context of NVML transactions. These opcodes are: `astore`, `aastore`, `bastore`, `castore`, `fastore`, `iastore`, `lastore`, `dastore`, `new`, `newarray`, `anewarray`, `multianewarray`, `putstatic`, and `putfield`. A description of each of these opcodes can be read at the Java Language Specification.[30]

The previous paragraph explained how JaphaVM ensures consistency at opcode granularity, but we still must enable programmers to define transactional behavior at application level. As described in Section 3.3, our design accomplishes that by using `synchronized` methods and blocks. In JaphaVM, we have modified the `monitorenter` and `monitorexit` opcodes in such a way that, when `monitorenter` is called, it starts a new NVML transaction. NVML transactions can be nested, so all subsequent opcodes will execute in the context of this transaction. Conversely, `monitorexit` completes the corresponding transaction. In the context of JaphaVM, we call those *user transactions*, as their scope is determined by the application programmer.

If any of the opcodes that change heap data are invoked outside the scope of an ongoing user transaction, each will begin a fine-grained transaction with the scope of the opcode execution. In the context of JaphaVM, we call those *automatic transactions*, as they are triggered internally by the JVM to define consistency points when modifying the heap.

Garbage collection. As GC modifies the persistent heap state, it must happen in the context of an NVML transaction. We have modified the `gc0` function responsible for GC, which is called in two situations: (1) during a synchronous GC, triggered by the VM running out of heap space to satisfy an allocation request, in which case the GC will happen inside a transaction that is already started; and (2) during an asynchronous GC, invoked by a specialized thread that triggers a GC periodically when the VM is idle. In JaphaVM, asynchronous GCs are only triggered when there are no outstanding transactions, to avoid isolation problems.

Handling external state. As described in Section 3.4, one of the challenges of OP is how to handle references to state that is external to the system. An example of that are file descriptors for `stdin`, `stdout`, and `stderr`. In GNU

```
/**
 * This class registers OPResumeListener objects for execution
 * upon JVM re-initialization.
 */
public class OPRuntime {

        /**
         * Adds a new listener.
         */
        public static void addListener(OPResumeListener listener);

        /**
         * Adds a new listener to a class that has a
     * static resume() method.
         */
        public static void addStaticListener(Class<?> clazz);

        /**
         * Method called by the JVM runtime to re-initialize
     * OPResumeListener objects.
         */
        public static void resumeAllListeners();

}
```

FIGURE 7 OPRuntime class

Classpath, these file descriptors are opened by native code that is invoked from the static initializer of the Java library class `gnu.java.nio.FileChannelImpl`. The Java Language Specification[30] determines that a static initializer is executed when the class is initialized, which takes place just once, when the class is originally loaded. However, we need to open the console file descriptors every time the VM is executed. None of the existing Java language abstractions provides a way to express this.

We have taken the approach used previously for PEVM[17] and PJama,[16] exposing to developers an API for making classes and objects *resumable*, ie, providing an interface for specifying methods to be executed every time the VM execution is resumed. To be resumable, an object must implement the interface OPResumeListener, which defines a `void resume()` method and be registered by the `OPRuntime.addListener()` class method. To execute resume code at the class scope, the class must implement a static `resume()` method and be registered by the `OPRuntime.addStaticListener()` class method.

We have implemented the `OPRuntime` class as part of GNU Classpath (its interface is depicted on Figure 7). To get the `resume()` methods to be executed whenever the JVM resumes execution, JaphaVM invokes `OPRuntime.resumeAllListeners()` just before the execution of the application's `main()` method.

Finally, we applied it to reopen the console file descriptors, modifying `gnu.java.nio.FileChannelImpl` to reinitialize them inside a static `resume()` method that is invoked in two occasions: (1) by its static initializer and (2) by `OPRuntime.resumeAllListeners()`, since it is registered using `OPRuntime.addStaticListener()` by the static initializer.

With this approach, not only can JaphaVM automatically reallocate its internal resources (such as the console file descriptors) but also is available for Java programmers to manage their own application-level resources.

One downside is that this mechanism violates OP's principle of *persistence independence*, as the programmer is required to manually implement resume behavior when holding references to objects that represent volatile external state, such as files and sockets. However, there seems to be no obvious way to solve this transparently.[1] On the other hand, it requires small programmer involvement when compared to traditional file-oriented or database-oriented approaches.

4.5 | Prototype limitations

As will be shown in next sections, JaphaVM has been successfully used to execute a wide range of complex programs and functions as a vehicle for both performance and development complexity experiments, showing the benefits of OP's

principles in NVM architectures. Still, as is the case with any engineering artifact, the current version of the JaphaVM prototype has the following limitations, which we plan to address in future versions:

- Interpreter inlining: Support for interpreter inlining (code-copying JIT) is not yet implemented in JaphaVM.
- Heap size: The heap size is currently fixed.
- Heap address relocation: Internal pointers to heap objects use fixed memory addresses in the current version, so the persistent heap must be always mapped to the same virtual address range.
- Data versus execution persistence: The current version supports only data persistence (see Section 3.2).
- Type evolution: JaphaVM currently does not support type evolution.

Note that the baseline JamVM version supports interpreter inlining and heap resizing, although JaphaVM currently does not. The other limitations refer to persistent heaps and thus are only applicable to JaphaVM.

5 | JAPHAVM EVALUATION

To assess the effectiveness of our design, we verified our prototype implementation along two axes:

1. *Execution performance*. Verify the relative performance of selected programs using orthogonally persistent data in PM compared to traditional implementations storing data in either files and databases.
2. *Development complexity*. Verify if the proposed design reduces the complexity of programs that rely on persistent data. Less complexity leads to reducing development effort and improved quality.[35,36]

To assess both development effort and execution performance, we have used two different workloads: (1) the OO7 benchmark and (2) a modified version of the Lucene search engine. Both are described in the next sections.

5.1 | Experimental setting

The experiments were executed on a 32-core machine with 244GB of main memory running Ubuntu Linux 12.04. All experiments were executed at least three times to identify variation and outliers. The results of the multiple executions were generally consistent, and the average results were used.

Since computers using NVM to implement PM are not yet available, we used DRAM to simulate PM, an approach employed by previous studies.[10,11] An area of 32 GB was reserved for a PM-aware file system: M1FS,[29] a configuration optimized for working directly with byte-addressable memory storage. JaphaVM was executed using a persistent heap with capacity of 8 GB, without considering metadata.

5.2 | OO7 benchmark

The OO7 benchmark[37] was originally aimed at object-oriented database management systems (OODBMS), and used in previous Java OP research.[17,18,38] This benchmark in a first step creates a database of objects combined in a tree-like hierarchy (including atomic and composite parts), whose nodes point to finer-grained objects with mutual references, forming a graph; as a second step, it performs multiple traversals through the graph (some of them modifying the contents of some nodes), and queries on the stored objects. Traversal 1 is a raw traversal of the complete hierarchy, traversal 2 is a traversal with updates, subdivided into (2a) update one atomic part per composite part; (2b) update every atomic part; and (2c) update each atomic part per composite part four times; finally, traversal 6 is a sparse traversal. The size of the OO7 database is configurable. For more details about the OO7 benchmark, please refer to its documentation.[39]

We have chosen the OO7 benchmark for a number of reasons. Since it was used in previous Java OP research, it permits a basic level of comparison. It models both read-intensive and write-intensive tasks on top of an in-memory graph, which is a common use case today. Finally, it permits a comparison of performance and development complexity between an OP implementation and a traditional relational database backend.

In our experiments, the relational database is mapped to Java objects by means of Hibernate, an object-relational mapping (ORM) framework. ORM is the preferred approach for most real-world, complex applications, as it removes from the programmer the burden of manually translating relational tables into objects and vice versa. In Section 5.3, we will consider another application that uses files as the backing store.

5.2.1 | Experiment description

Tests with OO7 were executed in the following distinct configurations:

1. *Disk-based DB*. Backed by a traditional disk-based database (PostgreSQL version 9.1), using the Hibernate framework version 3.2 for ORM.
2. *Memory-based DB* Backed by the H2 in-memory database (version 1.3.176), also using Hibernate.
3. *JaphaVM* — backed by a modified version of the benchmark using JaphaVMs persistence semantics; an in-memory data structure holds the data, Hibernate is not used. The NVML memory pool containing the persistent heap was created in M1FS, so as to simulate the behavior of a PM environment. JaphaVM was executed separately with three approaches for handling transactions (see Section 4.4):
    1. JaphaVM Automatic Tx: automatic Transactions, where each Java opcode modifying heap data is a self-contained transaction.
    2. JaphaVM User Tx: user Transactions, where the whole traversal is a single transaction within the bounds of a `synchronized` block.
    3. JaphaVm No Tx: no transactions, where there is no transactional overhead but also no fault tolerance.

We ran each configuration with three different database size presets (payloads): tiny, small, and medium. To understand the I/O profile of each scenario, during the execution of the benchmarks we collected statistics using the `/proc/PID/io` special file system interface.

5.2.2 | Analysis of results

Looking at the overall results of the OO7 benchmark, we observe that JaphaVM can create and traverse large object graphs with a performance two orders of magnitude better than database-backed scenarios using ORM layers. Figures 8A to 8C

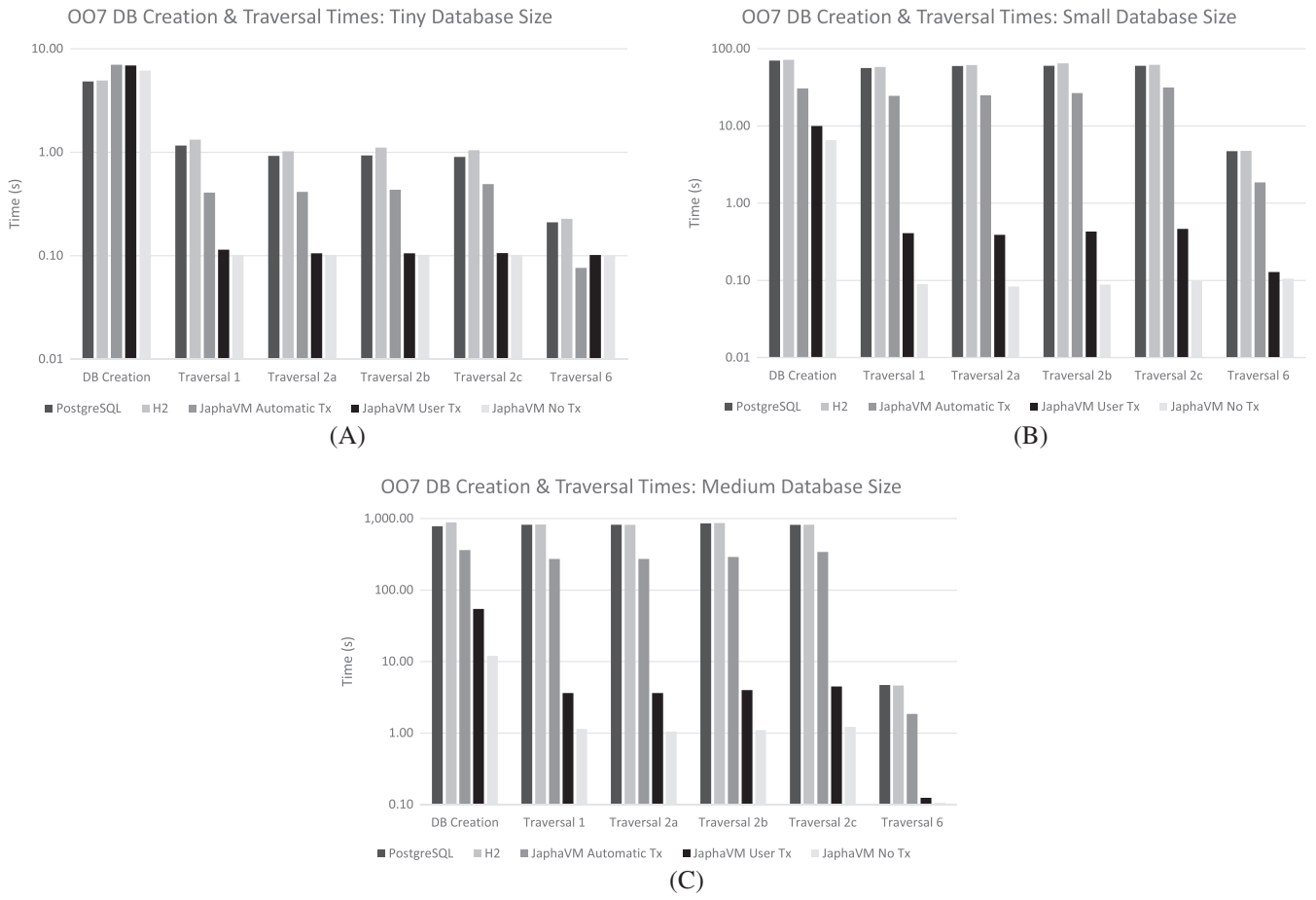
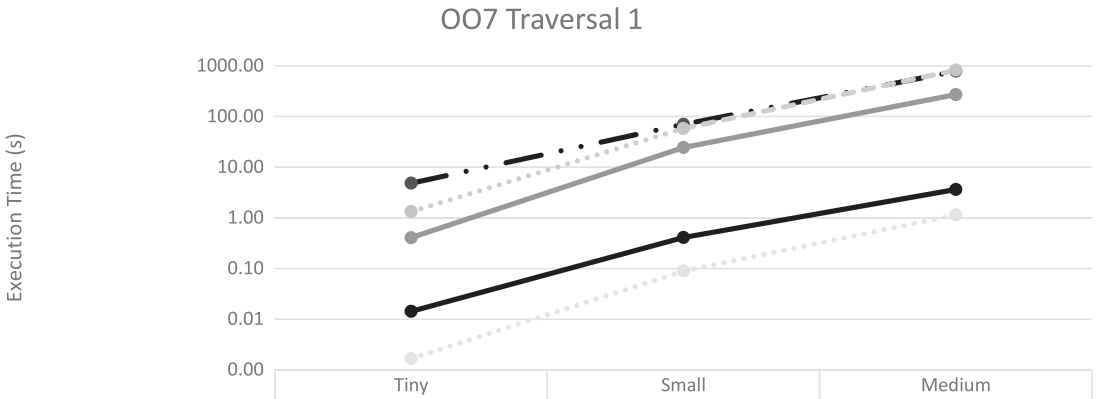


FIGURE 8 Comparison of execution times for each OO7 traversal across different database sizes (seconds, logarithmic scale). A, Comparison of execution times for tiny database size; B, Comparison of execution times for small database size; C, Comparison of execution times medium database size.



	Tiny	Small	Medium
• PostgreSQL	4.83	70.38	783.13
H2	1.32	58.21	823.09
JaphaVM Automatic Tx	0.41	24.60	272.14
JaphaVM User Tx	0.01	0.41	3.63
JaphaVM No Tx	0.00	0.09	1.14

FIGURE 9 Comparison of execution times for traversal 1 (seconds, logarithmic scale)

compare the execution time of creating the databases and each traversal for the different database sizes. We can observe in Figure 9, which highlights Traversal 1, that all execution scenarios have similar trends, despite the execution times being orders of magnitude different.

Tables 1 to 3 show I/O and execution counters for creating each database size, while Tables 4 to 6 show the same information for Traversal 1 on each database size. Read and written chars show the total number of characters read and written through system calls invoked by the program, irrespective if the operation was satisfied from the page cache or from the backing storage. Console I/O is also included in the read/written chars count. Read and write syscalls show the number of system calls invocations. They also list the number of voluntary and involuntary context switches.

As shown in Figure 8A, for the tiny database size JaphaVM takes about two seconds longer than PostgreSQL and H2 to create the database, due to memory pool initialization. However, for the creation of the small and medium databases (Figures 8B and 8C), JaphaVM scenarios execute one or two orders of magnitude faster than the relational databases. This is explained by the fact that JaphaVM requires less data movement and copy, which is confirmed by the data in Tables 1–3 (we show a single column for JaphaVM because I/O counters for all transactional scenarios were identical).

TABLE 1 I/O counters for *tiny* DB creation

Counter	PostgreSQL DB	H2 DB	JaphaVM
read chars	3 208 732	3 517 082	36 060
written chars	405 182	577 291	499
read syscalls	4126	6285	73
write syscalls	1602	3763	16
# voluntary context switches	138	123	13
# involuntary context switches	1	1	1

TABLE 2 I/O counters for *small* DB creation

Counter	PostgreSQL DB	H2 DB	JaphaVM
read chars	9 102 770	17 936 590	82 105
written chars	11 150 783	19 702 246	74 490
read syscalls	56 468	150 443	1174
write syscalls	53 958	147 928	1116
# voluntary context switches	1231	1225	1104
# involuntary context switches	1	1	1

TABLE 3 I/O counters for *medium* DB creation

Counter	PostgreSQL DB	H2 DB	JaphaVM
read chars	75 888 918	172 910 579	82 540
written chars	129 663 311	243 547 947	74 490
read syscalls	742 750	2 151 069	1174
write syscalls	741 264	2 148 581	1116
# voluntary context switches	1209	1238	1104
# involuntary context switches	1	1	1

TABLE 4 I/O counters for Traversal 1 on *tiny* DB

Counter	PostgreSQL DB	H2 DB	JaphaVM
read chars	3 590 297	4 393 436	29 636
written chars	539 701	1 047 596	504
read syscalls	3972	5827	70
write syscalls	1421	3320	16
# voluntary context switches	62	62	13
# unvoluntary context switches	1	1	1

TABLE 5 I/O counters for Traversal 1 on *small* DB

Counter	PostgreSQL DB	H2 DB	JaphaVM
read chars	34 465 024	78 717 157	29 652
written chars	33 360 993	60 359 612	520
read syscalls	87 846	190 519	51
write syscalls	85 294	335	11
# voluntary context switches	62	62	13
# unvoluntary context switches	1	1	1

The ORM layer used on the relational database scenarios creates many additional short-lived objects, and as consequence, the JVM spends more time doing GC. For the medium database size, all objects created by JaphaVM fit into the 8 GB heap, not requiring GC; while the relational database scenarios required between 56 and 68 GCs (15 to 16 of them also performing heap compactions), which accounted for 15.62% to 17.31% of the overall database creation time.

We also observe that JaphaVM execution has less context switches. This happens because it performs significantly less storage access, and thus is less blocked by I/O waits, which are a common cause for voluntary context switches.

As expected, the JaphaVM scenarios with less transactional overhead have better performance, ie, coarse-grained transactions (User Tx) have better performance than fine-grained transactions (Automatic Tx), and not supporting transactions at all has the best performance.

The relational database scenarios (PostgreSQL and H2) consistently take the longest time to execute traversal benchmarks. This is explained by the fact that these scenarios require ORM translation and access to storage, resulting in larger amount of data movement, which can be observed on Tables 4–6. Data movement is proportional to payload; the larger the database, more data movement is observed. PostgreSQL and H2 have very similar execution times, despite the former storing data on disk and the latter in memory. This happens because the experimental system has enough memory to keep the PostgreSQL database fully cached in memory. The trend for less context switches on JaphaVM is still observable, for the same reasons presented in the database creation analysis.

JaphaVM with automatic transactions (ie, a distinct transaction for each individual Java opcode modifying persistent heap data) executes the same traversals up to three times faster than PostgreSQL and H2, as it requires significantly less computational effort than the ORM scenarios. These results are achieved not only because JaphaVM runs completely in memory, but also because its software stack is shorter and optimized for PM. These effects can be observed in the small amount of data being read/written from the I/O subsystem. Its advantage is even more apparent with the larger database configurations, as the ratio between overhead and data volume for the Disk-based and Memory-based DB scenarios is higher. Additional gains can also be attributed to the lack of interprocess communication between the application and a database server in the JaphaVM scenario.

TABLE 6 I/O counters for Traversal 1 on *medium* DB

Counter	PostgreSQL DB	H2 DB	JaphaVM
read chars	384 046 805	803 662 022	86 127
written chars	308 148 130	553 984 993	75 953
read syscalls	800 981	1 763 481	1173
write syscalls	798 418	1 760 978	1180
# voluntary context switches	66	66	13
# unvoluntary context switches	1	1	1

JaphaVM with a single user transaction executes two orders of magnitude faster than the ORM scenarios, and almost one order of magnitude faster than JaphaVM with automatic transactions. In the latter, the great number of fine-grained NVML transactions impose a considerable overhead, as each single transaction needs to issue memory fences and cache drains to ensure that changes to the memory pool have reached the persistence domain upon transaction commit. By using a single coarse-grained transaction instead of multiple fine-grained ones, performance is significantly improved.

Results for JaphaVM with no transactions are listed as reference, as it cannot be practically used due to its lack of fault tolerance. We observe that this scenario executes about 30% faster than JaphaVM with a single user transaction, due to the absence of transactional overhead.

5.3 | Apache lucene

To evaluate JaphaVM using a real-world application, we have created a modified version of the Apache Lucene search engine library[40] using OP on top of JaphaVM. Lucene is a library written in Java that provides the ability of creating inverted indices of text documents, then allowing queries on what documents contains which terms. Lucene version 3.6.2 was used as baseline.

The original version of Lucene stores the inverted index in a set of segment files. Our "Lucene-OP" version uses instead a `java.util.HashMap` in the persistent heap, where each key is a term and each value has the type `java.util.HashSet` referencing the documents that contain that term (see Figure 10). Other than that, both versions perform the same tasks, such as tokenization, stop word filtering, stemming, and querying.

We have decided to experiment with a search engine since it is a critical, well-known real-world task, performed both in personal computers (eg, email search) and huge parallel machines (eg, web search), which depends critically on persistent data that can be easily held either in memory or in storage. We have chosen Lucene because it is the de facto standard library for text indexing and searching in Java. The fact that its original version uses files as backing store makes a good complement to our previous experiments with databases using OO7.

5.3.1 | Experiment description

We have compared Lucene and Lucene-OP executing three tasks:

1. *Indexing*. Index a corpus of text documents, namely Project Gutenberg's Aug 2003 compilation[41] containing 680 MB of texts, distributed across 595 documents containing 684 420 different terms.
2. *Single-term query*. Query all documents containing a single term *term1*, resulting in 523 hits.
3. *Double-term query* Query the documents containing both *term1* and *term2*, resulting in 49 hits.

Lucene-OP was executed with both user transactions and no transactional support. When executed with user transactions, during the indexing process, each transaction encompassed the complete indexing of a single document, and during querying each transaction encompassed the complete query.

5.3.2 | Analysis of results

Examining the overall results of the Lucene experiments, we confirm the observations from OO7 that graph traversal and pointer-chasing tasks are completed more than one order of magnitude faster using JaphaVM. However, we also observe that for tasks that are dominated by writes (such as the creation of an inverted index), writing to traditional files is potentially faster than their JaphaVM counterparts, due to the extra transactional overhead to keep the persistent heap failure-tolerant on the latter. This indicates different trade-offs for the two scenarios: when accessing storage devices,

```
public class InvertedIndex {

        protected Map<String, Set<Document>> invertedIndex;
        protected int numberOfDocs = 0;
        protected Map<Integer, Document> documents;

        public InvertedIndex() {
                invertedIndex = new HashMap<String, Set<Document>>();
        }

        public void addDocument(String term, Document doc) {
                // add to term/doc map
                Set<Document> documentSet = invertedIndex.get(term);
                if (documentSet == null) {
                        documentSet = new HashSet<Document>();
                        invertedIndex.put(term, documentSet);
                }
                documentSet.add(doc);
        }

        public Set<Document> getDocsForTerm(String term) {
                return invertedIndex.get(term);

        }

}
```

FIGURE 10 Code of Lucene OP inverted index

Time (ms)	Lucene (Baseline)	Lucene-OP User Tx	Lucene-OP No Tx
Indexing	1 124 932	2 491 539	1 052 148
Single-term Query	93	6.5	4
Double-term Query	113	7.5	4

TABLE 7 Lucene execution times (ms)

Time (ms)	Baseline	Lucene-OP
read chars	543 801 327	426 434 910
written chars	229 084 470	62 716
read syscalls	51 784	27 715
write syscalls	15 334	1209
# voluntary context switches	598	601
# unvoluntary context switches	987	355

TABLE 8 Lucene Indexing I/O counters

writes typically perform better than reads; however, when accessing PM with journaled transactions, writes are more expensive than reads.

Table 7 shows the execution time (in milliseconds) of each scenario using Lucene and Lucene-OP, the latter divided into user transactions or no transactions. We observe that the indexing time of Lucene-OP without transactions is 7% shorter than the original Lucene version. However, when transactions are introduced, Lucene-OP takes 55% longer than the baseline. This can be explained by the fact that both Lucene and Lucene-OP undergo a similar effort to read the input files (as we can infer from Table 8), but the index writing effort is directed either to files (Lucene) or to memory (Lucene-OP); writing to files is typically a fast operation (data are written to a buffer, and actual writes to disk are deferred), while writing to PM with transactional support carries additional overhead, as previously discussed in Section 5.2.2.

However, once data are committed to PM, queries become much faster than their file I/O counterparts. Looking at query execution times on Table 7, we observe that they are more than one order of magnitude faster than the baseline

TABLE 9 Lucene single-term query I/O counters

Time (ms)	Baseline	Lucene-OP
read chars	1 417 374	118 902
written chars	49 711	49 762
read syscalls	1972	640
write syscalls	1058	1060
# voluntary context switches	527	527
# unvoluntary context switches	1079	1081

TABLE 10 Lucene double-term query I/O counters

Time (ms)	Baseline	Lucene-OP
read chars	800 702	96 617
written chars	5141	5192
read syscalls	869	166
write syscalls	110	112
# voluntary context switches	53	54
# unvoluntary context switches	131	133

version. We can also observe that since queries are read-most operations (as confirmed by the I/O counters on Tables 9 and 10), the transactional overhead has less impact than on the indexing task.

5.4 | Development complexity

It is a known fact that less complexity leads to reduced software development and maintenance effort, and improved quality.[35,36] For this reason, it is desirable to keep programs as simple as possible, while ensuring that they perform their tasks correctly and efficiently; the advent of automatic memory management is an example of technique that was adopted with that purpose. Program complexity can be measured using diverse metrics, including lines of code, number of elements (such as classes, attributes, methods, etc) and class coupling, among others.

One of the main motivations for the OP approach is to remove from the programmer the burden of explicitly dealing with persistence, thus reducing program complexity. In the early 80s, Atkinson et al[2] stated in the founding paper of OP that about 30% of the lines of any program with a considerable amount of code would be concerned with transferring data to and from files or a DBMS. In a later study[14] they added that not only the program length increased complexity, but also the use of multiple application and system building mechanisms, such as databases, mapping frameworks and communication systems. They stated that this environmental complexity distracts the programmer from the task in hand, forcing them to concentrate on mastering the multiplicity of programming systems rather than the application being developed.

We have not identified recent studies quantifying the impact of persistence-specific code on program complexity, but our experience porting OO7 and Lucene to OP confirmed the expectation that they require less lines of code, components, dependencies, and overall development effort than their database- or file-oriented counterparts. We did not perform an extensive complexity evaluation, but we have gathered some simple metrics from these examples that provide initial insights on this subject.

We have compared both OO7 and Lucene baselines against their OP versions. Three simple complexity metrics were collected for each implementation: logical lines of code (LOC), number of classes, and number of methods. The OO7 results are listed on Table 11 and Lucene's on Table 12.

TABLE 11 Development complexity for OO7 scenarios

Metric	OO7 JaphaVM	OO7 DB	Hibernate Framework
LOC	987	1217	102 165
# Classes	18	19	1340
# Methods	163	172	12 281

TABLE 12 Development complexity for Lucene scenarios

Metric	Lucene-OP	Lucene
LOC	1067	8314
# Classes	79	314
# Methods	247	1513

When retrofitting OO7 for using JaphaVM's persistence semantics, we leveraged the existing classes implemented by the benchmark (Modules and Assemblies). Most of the work consisted of creating specialized subclasses that removed the ORM mapping from these classes to the database. As a result, the OO7 version using JaphaVM is slightly less complex than the traditional version using relational databases. However, the traditional version uses the Hibernate framework, which has a number of lines of code more than a hundred times larger than the OO7 benchmark itself, and a number of classes and methods about 75 times larger. The use of an ORM makes development simpler than manually programming the application interactions with the database, but is still reasonably more complex than handling persistence orthogonally, requiring from the programmer to master the ORM framework, and potentially additional runtime components.

Lucene-OP leverages many unmodified classes from the baseline implementation, especially for doing tokenization, stemming, and filtering. However, the code for both storing and querying inverted indices is significantly simpler on Lucene-OP. Table 12 shows that it requires seven times less LOCs, four times less classes and six times less methods to perform the same tasks.

6 | RELATED WORK

Previous designs for Java with OP were offered by PJama,[16] PEVM,[17] ANU OPJ,[18] Merpati,[19] and aspect-oriented programming implementations.[20,21] Differently from JaphaVM, none of these systems was designed for PM, so they still have to manage and move data across memory and storage, with all the entailed complexity and performance implications. On the other hand, the recent proposed Persistent Collections for Java (PCJ)[42] and Espresso[43] offer NVM support for Java, but at cost of providing a programming model to enable users to deal with persistent data.

Several recent studies proposed programming interfaces to manipulate data in PM in C and C++: Mnemosyne,[8] NV-Heaps,[9] Heapo,[11] Atlas,[10] and NVML.[12] Similarly to JaphaVM, most of these programming interfaces rely in some form of persistence by reachability. However, as they are designed for languages with manual memory management, special memory allocators, and/or persistence roots must be defined. Since JaphaVM is designed for a managed runtime language, it leverages the persistence roots already used by Java's automatic memory management. We used NVML as a lower-level layer for JaphaVM's failure-tolerant implementation, and others among the aforementioned C/C++ programming interfaces could have been used as well.

To the best of our knowledge, JaphaVM is the first JVM adherent to OP principles specifically designed to take advance of NVM architectures.

7 | CONCLUSION

The concept of OP was proposed in the early work of Atkinson et al,[2] advocating that, from a programmer's standpoint, there should be no differences in the way that short-term and long-term data are manipulated. In other words, persistence should be an *orthogonalproperty of data*, independent of data type and the way in which data is handled. Programmers should focus on the core aspects of their applications, whereas the runtime environment would take care of managing the longevity of data. The motivation behind this was to avoid moving data back and forth between these different layers and representations, what compromises both programming, and execution efficiency.

During the 1980s and 1990s, this concept was explored in several research initiatives, including programming languages, operating systems, and object-oriented databases.[1,3] However, the underlying implementations still had to cope with the complexity of moving data across memory and storage. Today, recent NVM technologies, such as resistive RAM and phase-change memory, allow main memory and storage to be collapsed into a single layer of PM, opening the way for more efficient programming abstractions for handling persistence, like OP.

In this work, we have revisited OP concepts in the context of NVM architectures and proposed a persistent heap design for languages with automatic memory management. We demonstrated how it can significantly increase programmer

and execution efficiency, removing the impedance mismatch of crossing semantic boundaries. To validate and explore the benefits of the OP concepts in practice, we presented JaphaVM, an implementation of the proposed design based on JamVM, an open-source JVM. JaphaVM's programming interface is inspired by previous research on OP JVMs, but its design leverages PM, resulting in a simpler implementation with better performance. It uses an original combination of OP, PM programming, persistence by reachability, and lock-based failure-atomic transactions, a concept that can be applied for other managed runtime languages, such as Python, Ruby, and JavaScript.

We have also described our prototype implementation and evaluated its advantages over traditional persistence approaches, such as relational databases and files, considering both execution performance and programming complexity. In our experimental results using benchmarks and real-world applications, JaphaVM in most cases executed the same operations between one and two orders of magnitude faster than database- and file-based implementations while requiring significantly less lines of code.

The current version of JaphaVM presents some shortcomings, such as lack of type evolution and execution persistence, which we plan to address in future research. However, it provides a good starting point for evaluating the advantages that PM is expected to bring to Java and other managed runtime applications: programs that are easier to write and maintain and have significantly superior performance.

ACKNOWLEDGEMENTS

The authors would like to thank Maiki Buffet, Gabriel Chiele, Natan Facchin, Rafael Furtado, Pedro Gyrão, Cristovam Lage, Matheus Alves, and Ícaro Raupp Henrique from the HP/PUCRS/LIS lab for their contributions to the implementation of JaphaVM. The authors would also like to thank Dejan S. Milojicic, Susan Spence, and Haris Volos from Hewlett-Packard Labs for reviewing early drafts of this paper. This research was supported in part by HP Inc and Hewlett-Packard Enterprise.

ORCID

Taciano D. Perez https://orcid.org/0000-0002-5256-0769
Marcelo V. Neves https://orcid.org/0000-0001-5996-3061
Diego Medaglia https://orcid.org/0000-0001-9715-6041
Pedro H. G. Monteiro https://orcid.org/0000-0002-5214-0422
César A. F. De Rose https://orcid.org/0000-0003-0070-0157

REFERENCES

1. Dearle A, Kirby GNC, Morrison R. Orthogonal persistence revisited. In: *Object Databases*. Berlin, Germany: Springer; 2010:1-22.
2. Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott PW, Morrison R. An approach to persistent programming. *Comput J*. 1983;26(4):360-365.
3. Dearle A, Hulse D. Operating system support for persistent systems: past, present and future. *Softw: Pract Exper*. 2000;30(4):295-324.
4. Lee BC, Ipek E, Mutlu O, Burger D. Architecting phase change memory as a scalable DRAM alternative. *ACM SIGARCH Comput Archit News*. 2009;37(3):2-13.
5. Qureshi MK, Srinivasan V, Rivers JA. Scalable high performance main memory system using phase-change memory technology. In: Proceedings of the 36th Annual International Symposium on Computer Architecture; 2009; Austin, TX.
6. Strukov DB, Snider GS, Stewart DR, Williams RS. The missing memristor found. *Nature*. 2008;453(7191):80-83.
7. SNIA. *NVM Programming Model (NPM) Version 1.1*. Technical Report. Colorado Springs, CO: Storage Networking Industry Association; 2015.
8. Volos H, Tack AJ, Swift MM. Mnemosyne: lightweight persistent memory. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems; 2011; Newport Beach, CA.
9. Coburn J, Caulfield AM, Akel A, et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems; 2011; Newport Beach, CA.
10. Chakrabarti DR, Boehm HJ, Bhandari K. Atlas: leveraging locks for non-volatile memory consistency. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications; 2014; Portland, OR.
11. Hwang T, Jung J, Won Y. Heapo: heap-based persistent object store. *ACM Trans Storage*. 2014;11(1):3.
12. Intel Corp. NVML library. 2016. http://pmem.io/nvml/
13. Lougher R. Jamvm - a compact java virtual machine. 2014. http://jamvm.sourceforge.net/
14. Atkinson M, Morrison R. Orthogonally persistent object systems. *VLDB J*. 1995;4(3):319-401.

15. Dearle A, Hulse D, Farkas A. Operating system support for Java. In: 1st International Workshop on Persistence and Java; 1996; Drymen, Scotland.
16. Atkinson M, Jordan M. *A Review of the Rationale and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform*. Technical Report. Mountain View, CA: Sun Microsystems, Inc; 2000.
17. Lewis BT, Mathiske B, Gafter N. *Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine*. Technical Report. Mountain View, CA: Sun Microsystems, Inc; 2000.
18. Marquez A, Blackburn S, Mercer G, Zigman JN. Implementing orthogonally persistent java. *Persistent object systems: Design, implementation, and use*. 2001:247-261.
19. Suezawa T. Persistent execution state of a Java virtual machine. In: Proceedings of the ACM 2000 Conference on Java Grande; 2000; San Francisco, CA.
20. Al-Mansari M, Hanenberg S, Unland R. Orthogonal persistence and AOP: a balancing act. In: Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software; 2007; Vancouver, Canada.
21. Pereira RHR, Perez-Schofield JBG. Orthogonal persistence in Java supported by aspect-oriented programming and reflection. Paper presented at: 6th Iberian Conference on Information Systems and Technologies (CISTI 2011); 2011; Chaves, Portugal.
22. Puglia GO, Zorzo AF, De Rose CAF, Perez TD, Milojicic D. Non-volatile memory file systems: a survey. *IEEE Access*. 2019;7:25836-25871.
23. Condit J, Nightingale EB, Frost C, et al. Better I/O through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles; 2009; Big Sky, MT.
24. Stornelli M. Protected and persistent ram filesystem. 2013. http://pramfs.sourceforge.net/
25. Dulloor SR, Kumar S, Keshavamurthy A, et al. System software for persistent memory. In: Proceedings of the 9th European Conference on Computer Systems; 2014; Amsterdam, The Netherlands.
26. Wilcox M. Support ext4 on NV-DIMMs. 2014. http://lwn.net/Articles/588218/
27. Wu X, Reddy AL. SCMFS: a file system for storage class memory. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis; 2011; Seattle, WA.
28. Xu J, Swanson S. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. Paper presented at: 14th USENIX Conference on File and Storage Technologies; 2016; Santa Clara, CA.
29. Plexistor SDM user guide. 2005. https://sourceforge.net/projects/oo7/
30. Gosling J. *The Java Language Specification*. Boston, MA: Addison-Wesley Professional; 2000.
31. Welc A, Hosking AL, Jagannathan S. Transparently reconciling transactions with locking for Java synchronization. Paper presented at: European Conference on Object-Oriented Programming; 2006; Nantes, France.
32. Farkas A, Dearle A. Changing persistent applications. In: Atkinson M, Maier D, Benzaken V, eds. *Persistent Object Systems*. London, UK: Springer; 1995:302-315
33. Perez T. JaphaVM code repository (GitHub). 2018.https://github.com/taciano-perez/JaphaVM
34. Free Software Foundation. GNU classpath. 2015. http://www.gnu.org/s/classpath/
35. Akiyama F. An example of software system debugging. In: Proceedings of IFIP Congress 1971; 1971; Ljubljana, Yugoslavia.
36. Kitchenham BA, Pickard LM, Linkman SJ. An evaluation of some design metrics. *Softw Eng J*. 1990;5:50-58.
37. Ibrahim A, Cook WR. Public implementation of oo7 benchmark. 2013. https://sourceforge.net/projects/oo7/
38. Ferreira P, Shapiro M. Larchant: persistence by reachability in distributed shared memory through garbage collection. In: Proceedings of 16th International Conference on Distributed Computing Systems; 1996; Hong Kong.
39. Carey MJ, DeWitt DJ, Naughton JF. The 007 benchmark. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data; 1993; Washington, DC.
40. Apache Software Foundation. Apache Lucene. 2016. https://lucene.apache.org/
41. Project Gutenberg. The August 2003 CD. 2003. https://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project
42. Intel Corp. Persistent collections for Java (PCJ). 2018. https://github.com/pmem/pcj.
43. Wu M, Zhao Z, Li H, et al. Espresso: brewing Java for more non-volatility with non-volatile memory. In: Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems; 2018; Williamsburg, VA.

How to cite this article: Perez TD, Neves MV, Medaglia D, Monteiro PHG, De Rose CAF. Orthogonal persistence in nonvolatile memory architectures: A persistent heap design and its implementation for a Java Virtual Machine. *Softw: Pract Exper*. 2020;50:368–387. https://doi.org/10.1002/spe.2781