PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

PERSISTENT MEMORY AND ORTHOGONAL PERSISTENCE: A PERSISTENT HEAP DESIGN AND ITS IMPLEMENTATION FOR THE JAVA VIRTUAL MACHINE

TACIANO DRECKMANN PEREZ

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. CÉSAR A. F. DE ROSE

Porto Alegre
2017

Ficha Catalográfica

P438p Perez, Taciano Dreckmann

Persistent Memory and Orthogonal Persistence : A Persistent Heap Design and its Implementation for the Java Virtual Machine / Taciano Dreckmann Perez . – 2017.

152 f.

Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Fonticielha De Rose.

1. Non-Volatile Memory. 2. Storage-Class Memory. 3. Persistent Memory. 4. Orthogonal Persistence. 5. Java Virtual Machine. I. De Rose, César Augusto Fonticielha. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Taciano Dreckmann Perez

Persistent Memory and Orthogonal Persistence: A persistent heap design and its implementation for the Java Virtual Machine

This Thesis has been submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science, of the Graduate Program in Computer Science, School of Computer Science of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on May 3rd, 2017.

COMMITTEE MEMBERS:

Prof. Dr. Dejan S. Milojicic (Hewlett Packard Labs)

Prof. Dr. Rodolfo Azevedo (IC/UNICAMP)

Prof. Dr. Avelino Zorzo (PPGCC/PUCRS)

Prof. Dr. César A. F. De Rose (PPGCC/PUCRS - Advisor)

To my family.

"The incoherence and complexity arising from utilizing many different programming mechanisms increases the cost both intellectually and mechanically of building even the simplest of systems."

Atkinson and Morrison, Orthogonally Persistent Object Systems [10]

"Disks are a hack, not a design feature.

The time and complexity penalty for using disks is so severe that nothing short of enormous cost-differential could compel us to rely on them. Disks do not make computers better, more powerful, faster, or easier to use. Instead, they make computers weaker, slower, and more complex. They are a compromise, a dilution of the solid-state architecture of digital computers. (...)

Wherever disk technology has left its mark on the design of our software, it has done so for implementation purposes only, and not in the service of users or any goal-directed design rationale."

Alan Cooper, About Face 2.0 [24]

ACKNOWLEDGEMENTS

This work has been developed in two interconnected research environments: PUCRS and Hewlett-Packard. I would like to thank both these institutions for providing me support, funding, equipment, and the opportunity to perform this research. I would also like to extend my gratitude to all my colleagues, managers, and teachers in both PUCRS and HP.

My advisor, Prof. César A. F. De Rose, guided me through the joys and challenges of academic research. He has been a counselor, a guide, and a friend, for which I am deeply grateful.

I would like to thank all the staff of the HP/PUCRS/LIS lab, and especially Maiki Buffet, Gabriel Chiele, Natan Facchin, Rafael Furtado, Pedro Gyrão, Cristovam Lage, Matheus Alves, and Ícaro Raupp Henrique for their contributions to the implementation of JaphaVM. It was great belonging to this team.

I owe a great debt to Pedro Garcez Monteiro, for his valuable contributions to JaphaVM, first at the LIS lab, and later as an HP colleague. Diego Rahn Medaglia and Marcelo S. Neves made important contributions to adapt the OO7 benchmark, create testing environments, and discuss experimental results. You guys are awesome.

Haris Volos and Susan Spence, from HP Labs, and Prof. Avelino Zorzo, from PUCRS, reviewed early drafts of this text and made many valuable suggestions. Dejan S. Milojicic, also from HP Labs, provided support and guidance in many ways. I am truly grateful for all their contributions.

I would also like to thank collectively all the pioneer researchers of orthogonal persistence, for seeing further into the future.

None of this would have been possible without my family: my wife Ana Paula da Fonseca, my daughter Valentina Velez and my mother-in-law Ana Maria Beati. They kept me sane and fed and warm and loved. I dearly love you all.

Since I was too young to remember, my mother, Neusa Marisa Elias Dreckmann, fanned the fires of my curiosity and taught me by example how to have the drive and resilience to turn visions into reality. I wouldn't be here if not for her, in many ways.

PERSISTENT MEMORY AND ORTHOGONAL PERSISTENCE: A PERSISTENT HEAP DESIGN AND ITS IMPLEMENTATION FOR THE JAVA VIRTUAL MACHINE

ABSTRACT

Current computer systems separate main memory from storage. Programming languages typically reflect this distinction using different representations for data in memory (e.g. data structures, objects) and storage (e.g. files, databases). Moving data back and forth between these different layers and representations compromise both programming and execution efficiency. Recent non-volatile memory technologies, such as Phase-Change Memory, Resistive RAM, and Magnetoresistive RAM make it possible to collapse main memory and storage into a single layer of persistent memory, opening the way for simpler and more efficient programming abstractions for handling persistence.

This Ph.D. thesis introduces a design for the runtime environment for languages with automatic memory management, based on an original combination of orthogonal persistence, persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions. Such design can significantly increase programming and execution efficiency, as in-memory data structures are transparently persistent, without the need for programmatic persistence handling, and removing the need for crossing semantic boundaries.

In order to validate and demonstrate the proposed concepts, this work also presents JaphaVM, the first Java Virtual Machine specifically designed for persistent memory. In 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.

Keywords: Non-Volatile Memory; Storage-Class Memory; Persistent Memory; Orthogonal Persistence; Java; Java Virtual Machine.

MEMÓRIA PERSISTENTE E PERSISTÊNCIA ORTOGONAL: UM PROJETO DE HEAP PERSISTENTE E SUA IMPLEMENTAçÃO PARA A MÁQUINA VIRTUAL JAVA

RESUMO

Sistemas computacionais da atualidade tradicionalmente separam memória e armazenamento. Linguagens de programação tipicamente refletem essa distinção usando diferentes representações para dados em memória (ex. estruturas de dados, objetos) e armazenamento (ex. arquivos, bancos de dados). A movimentação de dados entre esses dois níveis e representações, bidirecionalmente, compromete tanto a eficiência do programador quanto de execução dos programas. Tecnologias recentes de memoria não-volátil, tais como memória de transição de fase, resistiva e magnetoresistiva, possibilitam combinar memória principal e armazenamento em uma única entidade de memória persistente, abrindo caminho para abstrações mais eficientes para lidar com persistência de dados.

Essa tese de doutorado introduz uma abordagem de projeto para o ambiente de execução de linguagens com gerência automática de memória, baseado numa combinação original de persistência ortogonal, programação para memória persistente, persistência por alcance, e transações com atomicidade em caso de falha. Esta abordagem pode melhorar significativamente a produtividade do programador e a eficiência de execução dos programas, uma vez que estruturas de dados em memória passam a ser persistentes de forma transparente, sem a necessidade de programar explicitamente o armazenamento, e removendo a necessidade de cruzar fronteiras semânticas.

De forma a validar e demonstrar a abordagem proposta, esse trabalho também apresenta JaphaVM, a primeira Máquina Virtual Java especificamente projetada para memória persistente. Resultados experimentais usando benchmarks e aplicações reais demonstram que a JaphaVM, na maioria dos casos, executa as mesmas operações cerca de uma a duas ordens de magnitude mais rapidamente do que implementações equivalentes usando bancos de dados ou arquivos, e, ao mesmo tempo, requer significativamente menos linhas de código.

Palavras-chave: Memória Não-Volátil; Memória Persistente; Persistência Ortogonal; Java; Máquina Virtual Java.

LIST OF FIGURES

Figure 2.1 Napier88 architecture 35
Figure 2.2 SoftPM Architecture 39
Figure 3.1 Mnemosyne architecture 45
Figure 3.2 HEAPO API and system organization 51
Figure 4.1 Example of persistence by class attribute 66
Figure 4.2 Example of persistence by indirect class attribute reference 67
Figure 4.3 Example of persistence by stack reference 69
Figure 4.4 In-memory file system containing a heap file 70
Figure 5.1 Comparison of execution times for OO7 traversals of different db sizes (secs, log scale) 86
Figure 5.2 Curves with execution times for OO7 traversals across different db sizes (seconds, logarithmic scale) 87
Figure A.1 Overview of JamVM initialization steps 112
Figure A.2 Chunks organization inside heap 114
Figure A.3 Chunk header format 114
Figure A.4 Class organization in JamVM memory 115

LIST OF TABLES

Table 2.1 SoftPM API. 38
Table 3.1 Comparison of memory/storage technologies. 41
Table 3.2 Mnemosyne programming interface. 46
Table 5.1 Properties of OO7 database size presets. 85
Table 5.2 I/O counters for *tiny* DB creation. 88
Table 5.3 I/O counters for *small* DB creation. 88
Table 5.4 I/O counters for *medium* DB creation. 88
Table 5.5 I/O counters for Traversal 1 on *tiny* DB. 89
Table 5.6 I/O counters for Traversal 1 on *small* DB. 89
Table 5.7 I/O counters for Traversal 1 on *medium* DB. 89
Table 5.8 Lucene execution times (ms). 92
Table 5.9 Lucene Indexing I/O counters. 93
Table 5.10 Lucene single-term query I/O counters. 93
Table 5.11 Lucene double-term query I/O counters. 93
Table 5.12 Development complexity for OO7 scenarios. 94
Table 5.13 Development complexity for Lucene scenarios. 95

LIST OF ABBREVIATIONS

ACID	*Atomicity, Consistency, Isolation, Durability*
API	*Application Programming Interface*
AOP	*Aspect-Oriented Programming*
AOT	*Ahead of Time Compilation*
CPU	*Central Processing Unit*
DBMS	*Database Management System*
DIMM	*Dual In-Line Memory Module*
DRAM	*Dynamic Random-Access Memory*
GC	*Garbage Collection*
HDD	*Hard Disk Drive*
JEDEC	*Joint Electron Devices Engineering Council*
JNI	*Java Native Interface*
JIT	*Just In Time*
JVM	*Java Virtual Machine*
LOC	*Lines Of Code*
MDS	*Managed Data Structures*
MRAM	*Magnetoresistive Random-Access Memory*
NVM	*Non-Volatile Memory*
NVMe	*NVM Express*
NVML	*Non-Volatile Memory Library*
NV-DIMM	*Non-Volatile Dual In-line Memory Module*
OODBMS	*Object-Oriented Database Management System*
OOM	*Out-Of-Memory*
OP	*Orthogonal Persistence*
OS	*Operating System*
PCJ	*Persistent Collections for Java*
PCRAM	*Phase-Change Random-Access Memory*
PCM	*Phase-Change Memory*
PM	*Persistent Memory*
POS	*Persistent Object Store*
PRAM	*Phase-change Random-Access Memory*

RAM	*Random-Access Memory*
ReRAM	*Resistive Random-Access Memory*
SNIA	*Storage and Networking Industry Association*
RRAM	*Resistive Random-Access Memory*
SRAM	*Static Random-Access Memory*
SSD	*Solid-State Drive*
STT-MRAM	*Spin-Torque Transfer Magnetoresistive Random-Access Memory*
TML	*Tunneling Magneto-Resistance*
UML	*Unified Modeling Language*
VM	*Virtual Machine*

TABLE OF CONTENTS

1. INTRODUCTION 27
   1.1 Hypothesis and Research Questions . . . 28
   1.2 Thesis Organization . . . 28

2. ORTHOGONAL PERSISTENCE 31
   2.1 Orthogonal Persistence . . . 31
   2.2 Orthogonal Persistence and Programming Languages . . . 32
      2.2.1 PS-algol . . . 32
      2.2.2 Napier88 . . . 34
      2.2.3 Persistent Java . . . 35
      2.2.4 Java Card . . . 37
      2.2.5 SoftPM . . . 37
   2.3 Limitations and Criticism of Orthogonal Persistence . . . 38
   2.4 Conclusion . . . 39

3. NON-VOLATILE MEMORY AND PERSISTENT MEMORY 41
   3.1 Non-Volatile Memory . . . 41
      3.1.1 Phase-Change RAM (PCRAM) . . . 41
      3.1.2 Resistive RAM (ReRAM) . . . 42
      3.1.3 STT-MRAM . . . 42
   3.2 Persistent Memory . . . 42
   3.3 Operating System Support for Persistent Memory . . . 44
   3.4 Programming Languages Support for Persistent Memory: A Survey . . . 44
      3.4.1 Mnemosyne . . . 44
      3.4.2 NV-heaps . . . 47
      3.4.3 Atlas . . . 49
      3.4.4 Heapo . . . 51
      3.4.5 NVML . . . 52
      3.4.6 PyNVM . . . 55
      3.4.7 Persistent Collections for Java (PCJ) . . . 57
      3.4.8 Managed Data Structures (MDS) . . . 58
      3.4.9 Apache Mnemonic . . . 59
   3.5 Conclusion . . . 61

4. A DESIGN OF ORTHOGONALLY-PERSISTENT HEAPS FOR PERSISTENT MEMORY 63
4.1 Motivation . . . 63
4.2 System Assumptions . . . 64
4.3 Requirements . . . 64
4.4 Persistent Heap Adherent to Orthogonal Persistence Concepts . . . 64
4.5 Persistence by Class Attribute Reachability . . . 65
4.6 Persistence by Stack Reachability . . . 67
4.7 Storage and Management of Persistent Heaps . . . 68
4.7.1 Data Persistence . . . 70
4.7.2 Execution Persistence . . . 71
4.8 Consistency and Transactional Properties . . . 72
4.9 Challenges . . . 72
4.9.1 Sharing Data Across Different Programs and Programming Languages . . . 72
4.9.2 Type Evolution . . . 73
4.9.3 Consistency and Transactional Properties . . . 74
4.9.4 Persistent Bugs . . . 75
4.9.5 External State . . . 75
4.9.6 Security . . . 76
4.10 Conclusion . . . 76

5. JAPHAVM: AN ORTHOGONALLY-PERSISTENT JAVA VIRTUAL MACHINE DESIGNED FOR PERSISTENT MEMORY 77
5.1 JaphaVM Design . . . 77
5.1.1 Rationale . . . 77
5.1.2 JamVM and GNU Classpath . . . 77
5.1.3 Initial Design and its Limitations . . . 77
5.1.4 Final Design with Transactional Support . . . 78
5.1.5 Summary of JVM Modifications . . . 78
5.1.6 Prototype Limitations . . . 82
5.2 JaphaVM Evaluation . . . 83
5.2.1 Experimental Setting . . . 83
5.2.2 OO7 Benchmark . . . 83
5.2.3 Apache Lucene . . . 90
5.2.4 Development Complexity . . . 93
5.3 Evaluation Summary . . . 95
5.4 Conclusion . . . 96

6. RELATED WORK 97

7. CONCLUSION 99
7.1 Concluding Remarks . . . 100
7.2 Future Research . . . 101

Bibliography 103

A. APPENDIX: JAPHAVM IMPLEMENTATION DETAILS 111
A.1 JamVM Overview . . . 111
A.1.1 JamVM Initialization . . . 111
A.1.2 Internal Data Structures . . . 113
A.1.3 Threads and Thread Stacks . . . 118
A.1.4 Symbols Hash Table . . . 119
A.2 Changes Required to Implement JaphaVM . . . 122
A.2.1 Invocation Arguments . . . 122
A.2.2 Heap Persistence . . . 123
A.2.3 NVML Transactions . . . 125
A.2.4 NVML Memory Pool Initialization . . . 125
A.2.5 VM Initialization . . . 127
A.2.6 Persistent Heap Allocation . . . 128
A.2.7 Non-Heap Objects Persistence . . . 132
A.2.8 Hash Tables Persistence . . . 136
A.2.9 Context Recovery . . . 138
A.2.10 Java Opcodes . . . 138
A.2.11 Garbage Collection . . . 145
A.2.12 Handling External State: Console I/O . . . 148
A.2.13 JaphaVM Prototype Limitations . . . 152

1. INTRODUCTION

Existing computer systems traditionally separate main memory from storage. This distinction is not a design imperative, but rather 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 (e.g. data structures, objects) and in storage (e.g. files, databases). Moving and translating data back and forth between these different representations is inefficient both for programming (additional effort, complexity, maintenance challenges, and probability of defects) and execution (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 as the *Vietnam of Computer Science* [31].

Based on these observations, the concept of *orthogonal persistence (OP)* was proposed in the early 1980s [11]. 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 [29,31]. However, OP ended up not being widely adopted, as its underlying implementation still had to cope with the complexity of moving data between memory and storage, and the resulting performance consequences.

Recent byte-addressable, *non-volatile memory* (NVM) technologies such as Phase-Change RAM (PCRAM) [50,74], Resistive RAM (ReRAM) [79], and Magnetoresistive RAM (MRAM) [18,32,49] are expected to enable memory devices that are non-volatile, require low-energy and have density and latency closer to DRAM. These technologies make it possible to collapse main memory and storage into a single entity: *persistent memory (PM)*.

As consequence, many recent studies proposed *persistent memory programming* interfaces to manipulate data in byte-addressable, non-volatile memory [77]. Interfaces were proposed for languages with explicit memory management (such as C and C++) as early as 2011 [20,22,27,40,44,82]. Until very recently, little work had been carried out in order to explore persistent memory programming interfaces in languages with managed runtimes and automatic memory management, such as Java, Python, Ruby, and JavaScript, among others. Since 2016, works such as PyNVM [69], Persistent Collections for Java [28], Managed Data Structures [34], and Apache Mnemonic [5]) introduced specialized libraries for persistent memory in this category of programming languages.

This work introduces a design for the runtime environment of languages with automatic memory management based on an original combination of orthogonal persistence, persistent memory pro-

gramming, persistence by reachability, and lock-based failure-atomic transactions. It also presents JaphaVM, an implementation of the proposed design based on the JamVM [55] open-source Java Virtual Machine, in order to validate and demonstrate the presented concepts. To the best of our knowledge, our work is the first to propose an orthogonal persistence-oriented design for programming languages using persistent memory, and JaphaVM is the first JVM especially designed to take advantage of non-volatile memory technologies.

1.1 Hypothesis and Research Questions

The aim of this Ph.D. research is to investigate the hypothesis that *persistent memory can be used to solve the performance and complexity issues historically associated with orthogonal persistence support in programming languages with automatic memory management*. To guide this investigation, fundamental research questions associated with the hypothesis are defined as follows:

1. *Can we use a persistent heap on top of persistent memory to transparently store objects across multiple program executions?* The objective of this research question is to determine if it is possible to lower the complexity of implementing orthogonal persistence by using persistent memory as the single representation of data, avoiding dual memory vs. storage representations. The use of a persistent heap is expected not only to lower the design complexity, but also to improve execution performance.

2. *Is it possible to leverage automatic memory management mechanisms to determine the persistence lifecycle of ordinary objects when using a persistent heap?* The aim of this question is to determine if existing automatic memory management features (e.g., garbage collection) can be successfully used to transparently determine the persistence lifetime of ordinary objects in a persistent heap in persistent memory.

3. *How to automatically ensure the consistency of a persistent heap in the presence of failures?* The motivation for this research question comes from the fact that unexpected failures during the execution of a program directly accessing persistent memory can potentially leave data in inconsistent states. How can the runtime system address this problem instead of transferring it to the application programmer?

4. *What is the performance of an orthogonally-persistent system using a persistent heap relative to traditional storage approaches?* This objective of this research question is to identify if the proposed system has performance advantages over existing storage approaches (e.g. files, relational databases).

1.2 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** reviews the concepts of orthogonal persistence and summarizes the past efforts to incorporate them into programming languages, before the development of persistent memory.
- **Chapter 3** surveys the main non-volatile memory technologies and the current research efforts to design programming languages and operating system support for persistent memory.
- **Chapter 4** describes our proposal for revisiting orthogonal persistence concepts in the context of persistent memory, and presents a design for the runtime environment of languages with automatic memory management based on an original combination of OP, persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions. This chapter addresses research questions 1—3.
- **Chapter 5** introduces JaphaVM, an implementation of the proposed design, and evaluation results considering performance and ease-of-programming aspects. This chapter addresses research questions 1—4.
- **Chapter 6** discusses related work.
- **Chapter 7** summarizes the thesis and presents our concluding remarks. It restates the answers to the research questions and the main contributions, and presents possible directions for future work.
- **Appendix A** provides an in-depth description of JaphaVM's implementation.

2. ORTHOGONAL PERSISTENCE

As mentioned in the previous chapter, the cost of mapping data as represented in memory to either files or databases (and vice-versa) is known as *impedance mismatch*, and adds significant complexity to both software development and execution [31]. 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.

In this chapter, we recap the evolution of orthogonal persistence research and resulting concepts and challenges.

2.1 Orthogonal Persistence

In early 1980s, a research group led by Malcolm Atkinson proposed the concept of *orthogonal persistence* [11]. They observed that most programming languages provide separate abstractions for handling short-term and long-term data: short-term data are typically stored in memory, and manipulated using concepts such as arrays, records, sets, monitors and abstract data types (a.k.a objects), while long-term data are typically stored in file systems or database management systems (DBMS), and manipulated through file abstractions or DBMS models, such as the relational, hierarchical, network and functional models. The existence of two different views of data presents two problems, according to Atkinson et al. [11]:

1. In a typical program, about 30% of the code is concerned with transferring data to and from files or a DBMS. Is it necessary to spend effort in writing, maintaining and executing this code, and the quality of the application programs may be impaired by the mapping between the program's form of data and the form used for the long term storage medium.

2. Data type protection offered by programming language is often lost across this mapping. The structure that might have been exploited in a program to aid comprehension is neither apparent nor protected and is soon lost.

In order to solve these problems, they propose 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 manipulated, thus the term *orthogonal persistence*. During the decades of 1980 and 1990, this concept was explored in several research initiatives, including both programming languages and operating systems. The works of Alan Dearle et al. [29, 31] provide a good overview of this research, and the remaining of this chapter is largely based on these texts.

The key idea of orthogonally-persistent object systems is to support a uniform treatment of objects irrespective of their types by allowing values of all types to have whatever longevity is required. The benefits of orthogonal persistence can be summarized as [31]:

- improving programming productivity from simpler semantics;
- avoiding ad hoc arrangements for data translation and long-term data storage;
- providing type protection mechanisms over the whole environment;
- supporting incremental evolution; and
- automatically preserving referential integrity over the entire computational environment for the whole life-time of an application.

Atkinson and Morrison [10] identified three principles that, if applied, would yield orthogonal persistence:

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.

These principles were applied in different degrees in several experimental programming languages and operating systems. The next sections present a survey of these initiatives.

2.2 Orthogonal Persistence and Programming Languages

2.2.1 PS-algol

The first language to provide orthogonal persistence was PS-algol [11, 31]. It extended the existing S-algol language, providing *persistence by reachability*, where the identification of data to be persisted is made by reachability from a persistent root. Later, experience would show that this mechanism, also known as *transitive persistence*, is the only one that can be used to implement orthogonal persistence in languages that support references. It added the following abstractions to S-algol:

- **Tables** — associative stores (hash maps). A few functions are added to handle tables, such as `s.lookup`, which retrieves a value associated with a key in a table, and `s.enter`, which creates an association between a key and a value. Strictly speaking, tables are not directly related with persistence, but are useful abstractions that were not originally part of S-algol.

Listing 2.1 – PS-algol example of adding a record to a table

```
structure person(string name, phone; pntr addr)
structure address(int no ; string street, town)

let db = open.database("addr.db", "write")
if db is error.record
    do { write "Can't open database"; abort }

let table = s.lookup("addr.table", db)
let p = person("al", 3250,
        address(76, "North St", "St Andrews"))
s.enter("al", table, p)

commit
```

Listing 2.2 – PS-algol example of retrieving a record from a table

```
structure person(string name, phone; pntr addr)
structure person (string name, phone; pntr addr)

let db = open.database("addr.db", "read")
if db is error.record
    do { write "Can't open database"; abort }

let table = s.lookup("addr.table", db)
let p = s.lookup("al", table)
if p = nil then write "Person not known"
else write "phone number: ", p(phone)
```

- **Databases** — the roots of persistence in PS-algol, containing a pointer to a table. Everything referenced directly or indirectly by this table (including other tables) is identified as persistent. Databases can be dynamically created, and are manipulated through the `open.database` and `close.database` functions.

- **Transactions boundaries** — explicitly defined by the `commit` and `abort` functions.

In order to give a flavour of the language, two examples are provided. The first example (see Code Listing 2.1) opens a database called "addr.db" and places a person object into a table associated with the key "addr.table" found at its root. When `commit` is called, the persistence of all objects reachable from the root is ensured, including the updated table, the person object and the address object referenced by the person object.

The second example (Code Listing 2.2) opens the same database, retrieves the same person object and writes out their phone number:

A second version of PS-algol considered procedures as data objects, thus allowing both code and data to be persisted. PS-algol had no support for concurrency (other than at database level), which required additional programming effort.

Listing 2.3 – Napier88 program example

```
1 type person is structure (name, address : string)
2 let ps = PS()
3 project ps as X onto
4 person:
5     begin
6         X(name) := "Ronald Morrison"
7         X(address) := "St Andrews"
8     end
9 default: \{\}
```

2.2.2 Napier88

Napier88 [31,60] consisted not only of a language, but a complete self-contained persistent system, incorporating the whole language support within a strongly typed persistent store. It provides a pre-populated strongly typed stable store containing libraries for handling concurrency, distribution and user transactions, and a complete programming environment. Figure 2.1 displays the layered architecture of the Napier88 system.

Napier88 borrows from PS-algol the notion of persistence by reachability from a persistent root, but adds capabilities of dynamic type coercion, allowing the programmer to explicitly specify the type that a value being read from the persistent store will assume in the current context. This can be visualized in the example in Code Listing 2.3, that extracts a value from the persistent store, projects it into the `person` type and changes its name and address.

In the first line, the `person` type is declared. The `PS()` procedure, that returns the root of the persistent store, is then called and assigned to the variable `ps`. The `ps` variable is coerced to the type `person`. If this coercion is successful, a code block is executed to set the name and address of these types to constant values. If the coercion is not successful, the `default` option is executed (in this particular case, in case of failure the program does nothing). In order for the type coercion to be successful, the root of the persistent store must contain a pointer to data that can be cast to the `person` type.

The main additions that Napier88 introduces over PS-algol are listed below [31]:

- the infinite union type `any`, which facilitates partial and incremental specification of the structure of the data;
- the infinite union type `environment`, which provides dynamically extensible collections of bindings;
- parametric polymorphism, in a style similar to Java generics and C++ templates;
- existentially quantified abstract data types;
- a programming environment, with graphical windowing library, object browser, program editor and compiler, implemented as persistent objects withing the store;

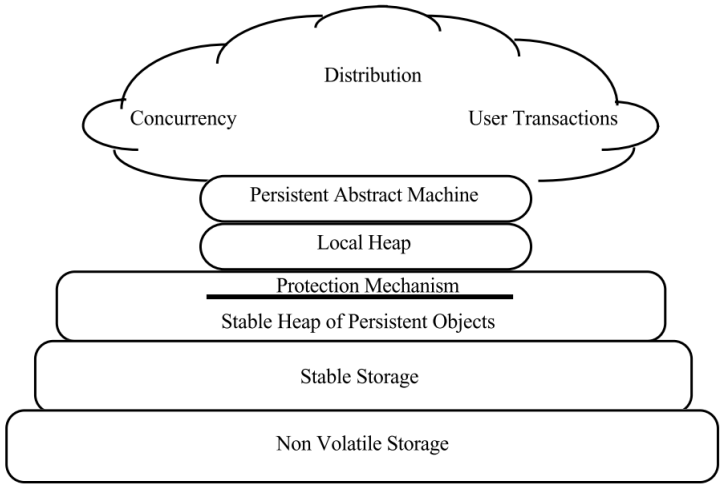


Figure 2.1 – Napier88 layered architecture. Extracted from [60].

- support for hyper-code, in which program source code can contain embedded direct references to extant objects;
- support for structural reflection, allowing a program to generate new program fragments and integrate those into itself.

2.2.3 Persistent Java

During the second half of the 1990s, several orthogonally-persistent versions of Java have been implemented.

PJama [9] started in 1995 as a collaboration between the Persistence and Distribution Research Group (PADRG) at Department of Computing Science at University of Glasgow and the Forest Research Group (FRG) at Sun Labs. It was actively developed and evolved until 2000, and served as reference for the Orthogonal Persistence for Java (OPJ) specification [46]. PJama implements persistence by reachability from the same roots used during garbage collection (the set of live threads and the static variables of the set of loaded classes) and from explicit persistent roots. The complete Java Virtual Machine (JVM) is checkpointed periodically, and its execution may be resumed in an arbitrary moment of the future from one of these checkpoints. The `Runtime` class is augmented with two methods: `checkpoint`, to explicitly generate a checkpoint and continue execution afterwards, and `suspend` to generate a checkpoint and then stop JVM execution. Since version 1.0, PJama started using its own custom persistent object store, called Sphere [72].

Code Listing 2.4 provides an example of a simple PJama program that increments an integer and explicitly triggers a checkpoint.

Some of the most notable limitations/challenges of the PJama implementation are:

1. Handling of bulk types — There is no query language for the manipulation of large object collections [8].

Listing 2.4 – Simple PJama program that increments an integer and explicitly triggers a checkpoint

```
1 public class Simple {
2
3   static int count = 0;
4
5   public static void main(String argv[]) {
6     System.out.println(count);
7     count++;
8     // explicitly trigger a checkpoint
9     org.opj.OPRuntime.checkpoint();
10  }
11
12 }
```

2. Conflict with traditional Java programming constructs — Java is based on some concepts that follow a traditional programming approach and are somewhat conflicting with orthogonal persistence principles, such as the `transient` keyword definition, the extensive usage of native code (via Java Native Interface - JNI) for windowing frameworks and handling of external resources, such as files and network-related structures [8].

3. Performance and complexity — relatively slow performance and high implementation complexity [51].

PJama influenced the design of PEVM [51], a JVM with native support for OP using a file-based persistent object store. PEVM is 10 faster than PJama Classic, largely due to its pointer swizzling strategy, which made it simple to modify its optimizing JIT while preserving almost all of the speed of the code it generates, and also due to its use of direct object pointers minimizing CPU overhead. Programming interfaces are very similar to PJama.

Another implementation of Persistent Java was made in the Grasshopper operating system [30]. Grasshopper supported persistence at operating system level, so no modifications were made to the Java language or virtual machine. All state was persisted in this implementation, but it relies on a special operating system designed for persistent memory.

ANU-OPJ [58] implemented persistence transparently through the dynamic manipulation of bytecodes, during class loading. All objects reachable from the roots during garbage collection were considered persistent and stored in the Shore storage manager. This approach results in full portability of Java programs, since it does not require a special JVM or Java compiler.

In the early 2000s, two articles [16, 80] proposed a form of OP supported by whole-system checkpointing of the JVM, its heap and threads. Both involve externalizing the JVM execution state to a file, which can be stored for later execution, or migrated over the network to a different node.

More recently, a number of papers [2, 63, 76] proposed Java OP implementations using Aspect-Oriented Programming (AOP). These implementations attempt to make the programmer oblivious to persistence concerns, which is handled as an aspect; persistence code is modularized and kept separate from application-specific code. Internally, a relational or object-oriented DBMS is used to

Listing 2.5 – Example of Java Card applet with persistent and transient data

```
1 import javacard.framework.*;
2
3 public class JavaCardApp extends Applet {
4
5  private JavaCardApp() {
6
7   this.persistent_array = new byte[ARRAY_SIZE];
8
9   this.transient_array = JCSystem.makeTransientByteArray(
10                  ARRAY_SIZE, JCSystem.CLEAR_ON_DESELECT);
11  }
12
13 }
```

store data.

All the systems above removed from the programmer the burden of explicitly translating data representations between memory and storage; however, the problem was transferred to the internal system implementation, which still had to cope with moving data between these different domains. This implied into high internal complexity and performance issues.

2.2.4 Java Card

The Java Card programming interface [61], used in JVM-based Smartcards, follows some elements from the Orthogonally Persistent approach. All objects are persistent by default, and stored in a persistent heap on top of EEPROM memory. In order to make an object transient (and thus mapped to RAM), it is necessary to declare it explicitly by using a static method `makeTransient...Array`, as shown in Code Listing 2.5. However, only arrays of primitive types can be made transient, which violates the principle of data type orthogonality. The lack of transient objects and the restriction on transient arrays requires the programmer to load and store values from persistent objects into transient arrays and vice versa.

An article from 1998 [61] proposes a transient environment composed by all objects referenced directly or indirectly from a transient root. It is analogous to the concept of persistence by reachability, defining transience by reachability in an environment where data is persistent by default. The condition is that references to transient objects are forbidden to be held in the persistent set, which must be enforced by the runtime environment.This scheme would be compliant with the principle of data type orthogonality, but was never adopted by the Java Card API specification.

2.2.5 SoftPM

SoftPM [40] is a programming abstraction for C providing orthogonal persistence for applications by means of containers that have a common persistence root.

Table 2.1 shows SoftPM's API for creating and managing containers. A container, or persis-

Table 2.1 – SoftPM API. Extracted from [40].

Function	Description
`int pCAlloc(int magic, int cSSize, void ** cStruct)`	*create a new container; returns a container identifier*
`int pCSetAttr(int cID, struct cattr * attr)`	*set container attributes; reports success or failure*
`struct cattr * pCGetAttr(int magic)`	*get attributes of an existing container; returns container attributes*
`void pPoint(int cID)`	*create a persistence point asynchronously*
`int pSync(int cID)`	*sync-commit outstanding persistence point I/Os; reports success or failure*
`int pCRestore(int magic, void ** cStruct)`	*restore a container; populates container struct, returns a container identifier*
`void pCFree(int cID)`	*free all in-memory container data*
`void pCDelete(int magic)`	*delete on-disk and in-memory container data*
`void pExclude(int cID, void * ptr)`	*do not follow pointer during container discovery*

tence root, is allocated using the `pCAlloc` function, but its contents are not made persistent at this point. Whenever the `pPoint` function is called, it creates a persistence point, i.e., all data reachable from the container will be made persistent. In subsequent executions of the program, the `pCRestore` function returns a pointer to a container previously created. Code Listing 2.6 describes the implementation of a persistent list. `pCAlloc` allocates a container and `pPoint` makes it persistent.

Listing 2.6 – SoftPM Example

```
1 struct c root
2 {
3   list t *l;
4 } *cr;
5
6 id = pCAlloc(m, sizeof(*cr), &cr);
7 cr->l = list head;
8 pPoint(id);
```

SoftPM's implementation consists of two main components: the Location Independent Memory Allocator (LIMA), and the Storage Optimized I/O Driver (SID). LIMA manages the container's persistent data as a collection of memory pages marked for persistence. When creating a persistence point, LIMA is responsible for identifying the graph of data structures referenced by the container and mark the ones which were modified, in order to be persisted. SID atomically commits container data to persistent storage, which can be disks, flash drives, network, or memcachedb. SoftPM's architecture is depicted on Figure 2.2.

2.3 Limitations and Criticism of Orthogonal Persistence

In the past, systems implementing orthogonal persistence handled internally the movement between memory and secondary storage, removing this burden from the programmer at the cost of increasing the internal complexity of these systems. The price paid in performance by doing so is one of the reasons that prevented orthogonally persistent systems from becoming a mainstream technology. As an example, Marquez et al. [57] compared the performance of different versions

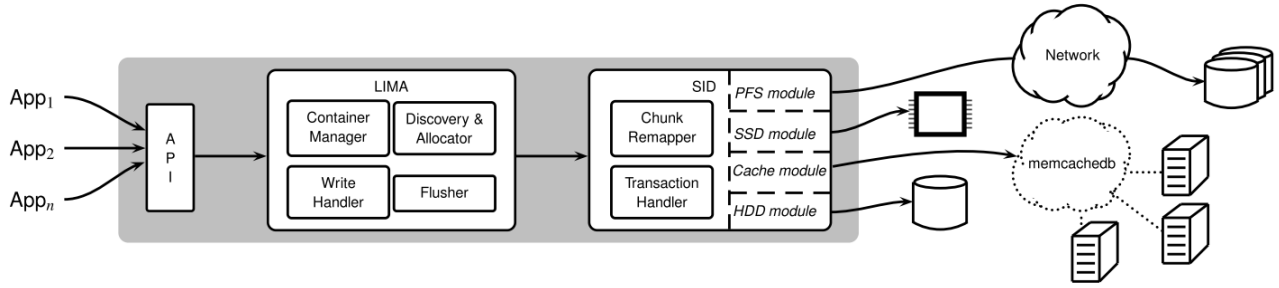


Figure 2.2 – SoftPM Architecture. Extracted from [40].

of PJama and ANU-OPJ using the OO7 benchmark [45], and in all scenarios the introduction of orthogonal persistence resulted in overheads of one or two orders of magnitude when compared to a standard JVM.

Several Persistent Operating Systems (POSs), i.e., OSs supporting abstractions to make persistence interfaces simpler, were explored. This body of research is summarized in [29] and [42]. Many of the challenges faced by such POSs were due to the need of managing movements of data between main memory and secondary storage. For instance, Multics, MONADS, and Clouds cannot guarantee consistency in the presence of unexpected failures, since changes to main memory may still not have been committed to secondary storage. Eumel/L3 solves this problem by taking periodic snapshots of the entire machine state, but incurs in performance and latency penalties in order to do so. Grasshopper faces overheads handling page faults due to secondary storage latency and several switches between the user-level/kernel boundary [42].

Cooper and Wise [25] argue that unrestricted orthogonal persistence, where a great number of objects live in a large memory addressing space and are shared by many different programs, can lead to bugs where programmers unintentionally modify data used by other programs, or keep references to objects no longer necessary. They propose a protection mechanism based on coarse-grained objects that contain many fine-grained objects, and apply protection and authorization mechanisms for the contained objects. The authors call this mechanism Type-Orthogonal Persistence, since any kind of object can still be persisted.

2.4 Conclusion

In this chapter, we have reviewed the evolution of orthogonal persistence research across the decades of 1980 and 1990, its results and limitations.

In the next chapter, we survey the existing efforts to design programming language support for persistent memory, which to the moment do not include OP-based interfaces.

3. NON-VOLATILE MEMORY AND PERSISTENT MEMORY

The previous chapter described how orthogonal persistence (OP) was created in the past to remove from the programmer the burden of explicitly managing data in both memory and storage, but how OP implementations still had to cope with the complexity of moving data between these two layers, and the resulting performance consequences. This chapter presents new non-volatile memory technologies that make it possible to collapse main memory and storage into a single entity: persistent memory (PM)

3.1 Non-Volatile Memory

There are several new non-volatile memory (NVM) technologies under research today to provide memory devices that are non-volatile, energy-efficient, with latencies close to DRAM and storage-class capacity. We have surveyed NVM technologies extensively in [64], and reproduce here a summary of the most mature technologies [49]: Phase-Change RAM (PCRAM), Resistive RAM (ReRAM), and Spin-Torque Transfer Magnetoresistive RAM (STT-MRAM).

Table 3.1 compares the main properties of traditional memory/storage technologies with PCRAM, ReRAM and STT-MRAM. Data was obtained from [18, 32, 49, 52, 59, 74].

3.1.1 Phase-Change RAM (PCRAM)

Phase-Change Random Access Memory (also called PCRAM, PRAM or PCM) is currently the most mature of the new memory technologies under research. It relies on some materials, called phase-change materials, that exist in two different phases with distinct properties: an amorphous phase, characterized by high electrical resistivity, and a crystalline phase, characterized by low electrical resistivity [75]. These two phases can be repeatedly and rapidly cycled by applying heat to the material [18, 75].

Table 3.1 – Comparison of memory/storage technologies.

	SRAM	**DRAM**	**Disk**	**NAND Flash**	**PCRAM**	**ReRAM**	**STT-MRAM**
Maturity	product	product	product	product	adv. dev.	early dev.	adv. dev.
Read latency	<10 ns	10-60 ns	8.5 ms	25 μs	48 ns	<10 ns	<10 ns
Write latency	<10 ns	10-60 ns	9.5 ms	200 μs	40-150 ns	10 ns	12.5 ns
Static power	yes	yes	no	no	no	no	no
Endurance	$>10^{15}$	$>10^{15}$	$>10^{15}$	$>10^{4}$	$>10^{8}$	$>10^{5}$	$>10^{15}$
Non-volatile	no	no	yes	yes	yes	yes	yes

3.1.2 Resistive RAM (ReRAM)

Despite the fact that PCRAM also uses resistance variances to store bit values, the term *Resistive RAM* (ReRAM) has been applied to a distinct set of technologies that explore the same phenomenon. Essentially these technologies fall into one of two categories [18]:

1. Insulator resistive memories: based on bipolar resistance switching properties of some metal oxides. The most important example is the *memristor* memory device, which will be further described in more detail.

2. Solid-electrolyte memories: based on solid-electrolyte containing mobile metal ions sandwiched between a cathode and an anode. Also known as Programmable Metallization Cell (PMC) or Conductive Bridge RAM (CBRAM).

There is a long list of ReRAM technologies [17, 18, 49, 88].

A memristor [88] is a two-terminal device whose resistance depends on the magnitude and polarity of the voltage applied to it and the length of time that voltage has been applied. When the voltage is turned off, the memristor remembers its most recent resistance until the next time it is turned on. The property of storing resistance values means that a memristor can be used as a nonvolatile memory [85].

3.1.3 STT-MRAM

Magnetoresistive RAM (MRAM), sometimes called Magnetic RAM, is a memory technology that explores a component called Magnetic Tunnel Junction (MTJ), consisting MTJ of two ferromagnetic layers separated by an oxide tunnel barrier layer. One of the ferromagnetic layers, called the reference layer, keeps its magnetic direction fixed, while the other, called the free layer, can have its direction changed by means of either a magnetic field or a polarized current. When both the reference layer and the free layer have the same direction, the resistance of the MTJ is low. If they have different directions, the resistance is high. This phenomenon is known as Tunneling Magneto-Resistance (TMR) [18, 32, 49].

Conventional MRAM (also called "toggle-mode" MRAM) uses a current induced magnetic field to switch the MTJ magnetization. The amplitude of the magnetic field must increase as the size of MTJ scales, which compromises MRAM scalability. Spin-Torque Transfer MRAM (STT-MRAM) technology tries to achieve better scalability by employing a different write mechanism based on spin polarization [53].

3.2 Persistent Memory

These novel memory technologies can potentially make a significant amount of NVM available to computer systems via memory bus. It would thus make it possible for main memory and secondary storage to be collapsed into a single entity: *persistent memory (PM)*. NVM would be available to

the operating system as byte-addressable, persistent memory that can be directly addressed by the processor in the same way that DRAM is today, but with the advantage of being non-volatile, or, in other words, persistent.

There are several advantages in using persistent memory instead of separating the memory hierarchy in volatile main memory and persistent storage. HDDs and SSDs are widely employed today not for any goal-directed design rationale, but for the fact that today's technology standards make them extremely cost-effective, and it has been so since the beginning of commercial computing. As Alan Cooper [24] pointed out, "Disks do not make computers better, more powerful, faster, or easier to use. Instead, they make computers weaker, slower, and more complex. (...) Wherever disk technology has left its mark on the design of our software, it has done so for implementation purposes only, and not in the service of users or any goal-directed design rationale."

But decades of separation between volatile main memory and persistent storage have left deep marks in computer design, including hardware, basic software and application software. If the emerging memory technologies fulfill their promise of enabling persistent memory, several layers of the hardware and software stack will be affected.

Changing hardware, operating systems and programming languages internal implementations can be challenging for computer scientists and engineers, but they potentially have a mild impact on application software when compared to conceptual changes that affect the interfaces exposed by programming languages. Nonetheless, we believe that in order to reap the major rewards potentially offered by persistent main memory, it will be necessary to change the way application developers approach programming. Current programming interfaces have separate abstractions for handling memory (data structures) and secondary storage (files, databases). A good deal of development effort is directed towards moving data between these two layers; a seminal study estimates around 30% [11].

In our M.Sc. dissertation [64], we evaluated the impacts on latency and energy of a computer system with persistent memory. We simulated scenarios of systems using different technologies for main memory (DRAM, PCRAM and Memristor) using a full-system architectural simulator (Virtutech Simics [56]), and compared the execution of workloads using both a typical scenario using the file system as storage backend and an API specific for persistent memory in C/C++ (Mnemosyne [82]). Additional results were also published in [66] and [67]. During that study, our own development effort using an API designed for persistent memory produced less than half lines of code and was more than 4x faster to develop than using traditional file-oriented APIs. Our conclusion was that in order to leverage persistent memory to simplify the task of designing, developing and maintaining software, we need to change programming interfaces.

The next sections survey the current efforts towards designing operating systems and programming languages exploiting persistent memory.

3.3 Operating System Support for Persistent Memory

The emergence of new NVM technologies is reigniting the interest on OS support for persistent memory. In the 2013 Linux Foundation Collaboration Summit [26], the session "Preparing Linux for nonvolatile memory devices" proposed a three-step approach:

1. In a first moment, access NVM devices using traditional block drivers, by means of existing file systems. This step does not explore the disruptive potential of NVM but is a fast way of making it accessible.

2. In a second moment, adapt existing file systems to access NVM directly, in a more efficient way. This step ensures that file systems are designed for NVM, but keeps the traditional file system abstraction.

3. The final step is thinking about the creation of byte-level I/O APIs for new applications to use. This step will explore interfaces beyond the file system abstraction that may be enabled by NVM. It has the most disruptive potential, but requires applications to be modified to use such interfaces.

Bailey et al. [12] discuss additional changes to paging, virtual memory, reliability and security.

In the past few years, several file system designs were proposed addressing step 2: PRAMFS [78], BPFS [23], SCMFS [86], PMFS [33], Aerie [81], and Ext4/DAX [84]. They expose NVM using a file system interface, and perform significantly better than disk- or flash-based file systems since they bypass the page cache, and remove other steps rendered unnecessary by NVM, which can be directly accessed by the processor at byte granularity.

More recently, Microsoft announced persistent memory support starting in Windows Server 2016 and Windows 10, including native file systems [21].

3.4 Programming Languages Support for Persistent Memory: A Survey

Recent works proposed interfaces for manipulating data directly in persistent memory, such as Mnemosyne [82], NV-Heaps [22], Atlas [20], Heapo [44], NVML [27], PyNVM [69], PCJ [28], MDS [34], and Apache Mnemonic [5]. These interfaces greatly simplify data manipulation, avoiding the translation between memory data structures and a file (or database) format. However, they are not OP implementations, and still require specific interfaces and explicit manipulation of persistent data. In the next sections, we present an original survey of these proposals, discussing them in detail.

3.4.1 Mnemosyne

Mnemosyne [82] proposes an abstraction of *persistent memory* to be made available by operating systems to applications. This abstraction enables programmers to make in-memory data structures persistent without converting it to serialized formats. Direct access also reduces latency because

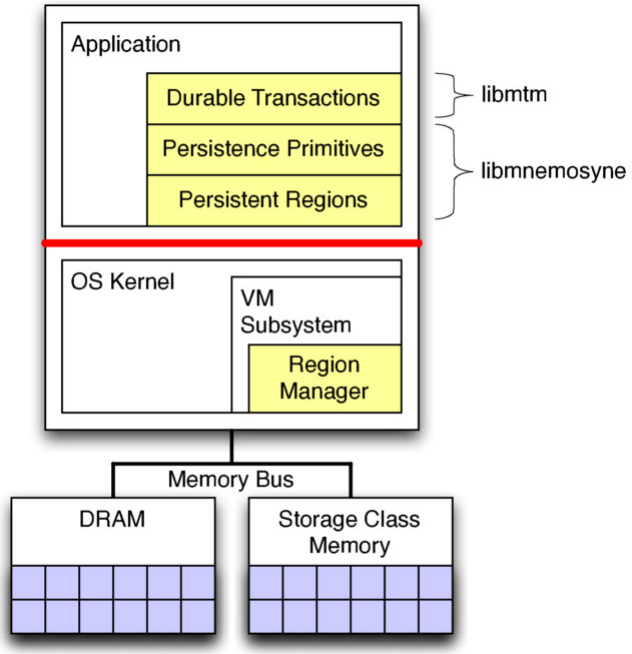


Figure 3.1 – Mnemosyne architecture. Extracted from [82].

it bypasses many software layers including system calls, file systems and device drivers. It uses transactional memory constructs in order to ensure consistency. Mnemosyne's goals are providing a simple interface for the programmer, ensure consistency across modifications and being compatible with existing commodity processors.

Mnemosyne provides a low-level programming interface, similar to C, for accessing persistent memory. It supports three main abstractions:

1. *Persistent memory regions* — segments of virtual memory mapped to non-volatile memory. Regions can be created automatically to hold variables labeled with the keyword `pstatic` or allocated dynamically. Persistent regions are implemented by swapping non-volatile memory pages into a backing file.

2. *Persistence primitives* — low-level operations that support consistently updating data.

3. *Durable memory transactions* — mechanism that enables consistent in-place updates of arbitrary data structures.

The Mnemosyne architecture can be seen at Figure 3.1.

Mnemosyne is originally implemented in Linux and extends the OS virtual memory system by adding a *region manager*, responsible for memory management in the non-volatile space address. A list of the virtual pages allocated in persistent memory is maintained in a *persistent mapping table*. The region manager reconstructs persistent regions when the OS boots. Persistent pages are mapped to backing files, using a variation of the `mmap()` system call.

Mnemosyne provides *persistence primitives* of distinct abstraction levels, as depicted in Table 3.2. At the lowest level, there are hardware primitives to allow storing and flushing data into persistent

Table 3.2 – Summary of Mnemosyne's programming interface. Extracted from [82].

Class	API	Description
H/W primitives	flush(addr) store(addr, val) wtstore(addr, val) fence()	Writes back and invalidates the cache line that contains the linear address *addr*. Writes value *val*. Writes value *val* to SCM. Prevents subsequent writes from completing before preceding writes.
Persistent regions	`pstatic` var pmap(addr, len, prot, flags) punmap(addr, len) type `persistent` * ptr	Allocates the variable *var* in the static region. Creates a dynamic region. Deletes part or all of a dynamic region. Declares the target of the pointer *ptr* as persistent.
Persistent heap	pmalloc(sz, ptr) pfree(ptr)	Sets *ptr* to point to a newly allocated persistent memory chunk of size *sz*. Deallocates the persistent memory chunk pointed by *ptr* and then nullifies *ptr*.
Log	log_create(flags, cbf) log_append(rec) log_flush() log_truncate()	Creates a log. Writes record *rec* by appending it at the end of the log. Blocks until all prior writes to the log reach SCM. Drops any records written to the log.
Durable transactions	atomic {...}	Atomically updates persistent state.

memory. At a higher level, variables can be automatically assigned to be persisted, both statically (through the `pstatic` declaration) and dynamically (being allocated/freed with `pmalloc`/`pfree`). At the highest level, all code inside an `atomic` block has transactional properties. Mnemosyne uses write-ahead redo logging to ensure the atomicity of transactions, and exposes its log API so application code can also make use of it for their own purposes.

Code Listing 3.1 is an example of using Mnemosyne with static variables. Every time the program executes, the value of the integer variable `flag` will switch between 0 and 1.

Listing 3.1 – Mnemosyne Example Using Static Variables

```
1  #include <stdio.h>
2  #include <mnemosyne.h>
3  #include <mtm.h>
4
5  MNEMOSYNE_PERSISTENT int flag = 0;
6
7  int main (int argc, char const *argv[]) {
8    printf("flag: %d", flag);
9    MNEMOSYNE_ATOMIC {
10   if (flag == 0) { flag = 1; }
11     else { flag = 0; }
12   }
13   printf(" --> %d\n", flag);
14     return 0;
15 }
```

A second example (Code Listing 3.2) shows a simple hash table being manipulated with dynamic variables.

Listing 3.2 – Mnemosyne Example Using Dynamic Variables

```
1  #include <mnemosyne.h>
```

```
2 #include <mtm.h>
3
4 MNEMOSYNE_PERSISTENT hash = NULL;
5
6 main() {
7   if (!hash) {
8     pmalloc(N*sizeof(*bucket), &hash);
9   }
10 }
11
12 update_hash(key, value) {
13   lock(mutex);
14   MNEMOSYNE_ATOMIC {
15     pmalloc(sizeof(*bucket), &bucket);
16     bucket->key = key;
17     bucket->value = value;
18     insert(hash, bucket);
19   }
20   unlock(mutex);
21 }
```

The performance of applications such as OpenLDAP and Tokyo Cabinet (a key-value store) using Mnemosyne's persistent memory increased by 35–117 percent when compared to Berkeley DB or flushing the data to disk [82]. In some scenarios, Tokyo Cabinet performance improved by a factor of 15.

The implementation of Mnemosyne consists of a pair of libraries and a small set of modifications to the Linux kernel for allocating and virtualizing non-volatile memory pages. It runs on conventional processors. Optionally, Makalu [14] provides an alternative memory allocator with improved performance.

The purpose of Mnemosyne is to reduce the cost of making data persistent, since current programs devote large bodies of code to handling file system and/or database access. On the other hand, Mnemosyne is proposed not as a replacement for files, but as a fast mechanism to store moderate amounts of data; thus, applications should still use files for interchanging data and for compatibility between program versions when internal data-structure layouts change.

3.4.2 NV-heaps

Non-volatile Memory Heaps (NV-heaps) [22] is a persistent object system implemented specifically for non-volatile memory technologies such as PCM and MRAM. It was designed concurrently with Mnemosyne. NV-heaps was designed to present the following properties:

1. **Pointer safety** — it prevents programmers, to the extent possible, from inadvertently corrupting their data structures by pointer misuse or memory allocation errors.

2. **Flexible ACID transactions** — NV-heaps supports multiple threads accessing common data, and provides consistency across system failures.

3. **Familiar interface** — the programming interface is similar to the familiar interface for programming volatile data.

4. **High performance** — it was designed to explore as much as possible the speed of the underlying persistent memory hardware.

5. **Scalability** — NV-heaps scale to datastructures up to many terabytes.

NV-heaps proposes the following abstractions:

- **NV-heaps** — non-volatile heaps that are managed as files by a memory-based ordinary file system (which provides naming, storage management and access control), and that can be mapped by the framework directly into the program space address.

- **Persistent objects** — objects that inherit from a superclass provided by the framework (`NVObject`), and can be persisted to NV-heaps.

- **Root pointer** — every NV-heap has a root pointer from which all objects reachable are persisted.

- **Smart pointers** — smart pointers are needed to reference persistent objects. NV-heaps uses these pointers to do reference counting and manage object locks. Two types of pointers are supported: V-to-NV (volatile references to non-volatile data) and NV-to-NV (non-volatile references to non-volatile data).

- **Transactions** — programmers explicitly define the boundaries of transactions with ACID properties.

NV-heaps is implemented as a C++ library under Linux. NV-heaps are created as regular files, and mapped to to the application's virtual address space using `mmap()`.

Code listing 3.3 exemplifies the removal of a value `k` from a persistent linked list.

Listing 3.3 – NV-Heaps Example

```
1 class NVList : public NVObject {
2   DECLARE_POINTER_TYPES(NVList);
3
4 public:
5   DECLARE_MEMBER(int, value);
6   DECLARE_PTR_MEMBER(NVList::NVPtr, next);
7 };
8
9 void remove(int k) {
10   NVHeap * nv = NVHOpen("foo.nvheap");
```

```
11   NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
12   AtomicBegin {
13     while(a->get_next() != NULL) {
14       if (a->get_next()->get_value() == k) {
15         a->set_next(a->get_next()->get_next());
16       }
17         a = a->get_next();
18     }
19   } AtomicEnd;
20 }
```

The `NVList` class extends `NVObject`, thus being persistent. The `DECLARE_POINTER_TYPES`, `DECLARE_MEMBER` and `DECLARE_PTR_MEMBER` macros declare the smart pointer types for NV-to-NV (`NVList::NV_Ptr`) and V-to-NV (`NVList::V_Ptr`) and declare two fields. The declarations generate private fields in the class and public accessor functions (e.g., `get_next()` and `set_next()` that provide access to data and perform logging and locking).

The program invokes `NVHOpen` to open an NV-heap and retrieves the root pointer. The block delimited by `AtomicBegin`/`AtomicEnd` atomically iterates through the nodes of the linked list and removes all that are equal to `k`.

3.4.3 Atlas

Atlas [20] is a system that leverages locks for ensuring consistency in non-volatile memory, adding durability to lock-based code. It is currently implemented in C.

Atlas defines containers called *persistent regions* for identifying persistent data. A persistent region is similar to memory-mapped file with associated metadata. It consists of an identifier and a list of virtual address ranges. Each persistent region has a persistence root, and all data contained by persistent regions that is reachable from their roots is considered implicitly persistent. Data in transient regions or not reachable from persistent region roots are considered transient. Code listing 3.4 shows an example of how to access persistent regions in Atlas.

Listing 3.4 – Accessing Persistent Regions in Atlas

```
1  pr = find_or_create_pr(name);
2  persistent_data = get_root_pointer(pr);
3  if (persistent_data) { A; // restart-only code }
4  else { B; set_root_pointer(...); // initialize }
5  C; // use persistent_data
```

Durable updates are stores to persistent memory that are in consistent state during subsequent executions of a program. The current Atlas implementation uses an undo log to ensure that a set of stores inside a *failure-atomic section* will be consistent and will respect atomicity. The boundaries of failure-atomic sections are defined by regular locks defined by the program. During compilation, the object code is instrumented to add instructions handling failure-atomic sections and log operations.

Code Listing 3.5 shows a simple alarm clock example using Atlas.

Listing 3.5 – Simple Alarm Clock Example Using Atlas

```
struct Alarm {int hr; int min; Alarm *next;};
mutex m;
// protects updates to Alarm structure PR alarm_pr;
main() {
  alarm_pr = find_or_create_pr("alarm_clock");
  Alarm *a = get_root_pointer(alarm_pr);
  if (a) print(a);
  // restart-only code
  // Actual code for the following is not shown.
  // Two threads are created. Thread 1 can set/
  // update an alarm by calling set_or_update_alarm.
  // Thread 2 runs thread_2 below to sound the
  // alarm when appropriate.
}
void set_or_update_alarm(int hr, int min) {
  m.lock();
  Alarm *a = get_root_pointer(alarm_pr);
  if (!a) {
    a = (Alarm*)pmalloc(sizeof(Alarm), alarm_pr);
    set_root_pointer(alarm_pr, a);
  }
  a->hr = hr; a->min = min; a->next = NULL;
  m.unlock();
}
void thread_2() {
  for (;;) {
    int cur_hr, cur_min;
    get_current_time(&cur_hr, &cur_min);
    m.lock();
    Alarm *a = get_root_pointer(alarm_pr);
    if (a && cur_hr == a->hr && cur_min == a->min){
      sound_alarm();
      // cancel alarm
      set_root_pointer(alarm_pr, NULL); pfree(a);
    }
    m.unlock();
    sleep_a_few_seconds();
  }
}
```

Optionally, Makalu [14] provides an alternative memory allocator with improved performance.

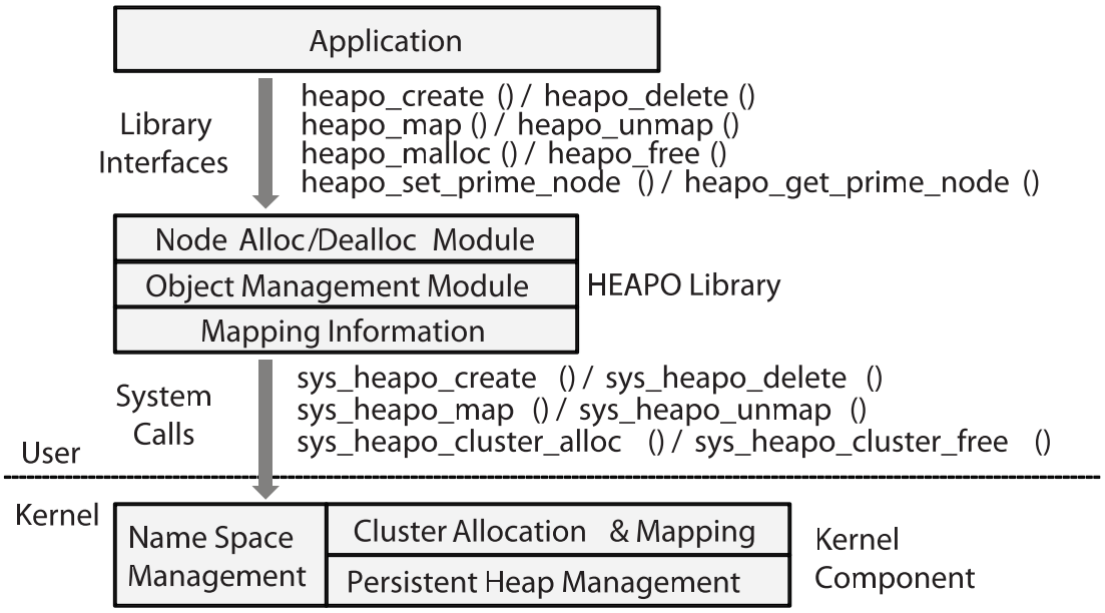


Figure 3.2 – HEAPO API and system organization. Extracted from [44].

3.4.4 Heapo

Heapo [44] is a persistent heap implementation for C.

Differently from Mnemosyne, NV-Heaps and Atlas, its implementation does not use memory-mapped files. HEAPO implements a native layer in the OS kernel, with its own metadata, heap layout, access control, name space, and other parameters. HEAPO reserves a fixed address range in the virtual address space for the persistent heap, and HEAPO uses the existing Linux buddy memory allocation algorithm [48] to manage persistent memory. It maintains page frame allocation information on the first page of persistent memory. It addresses crash-recovery by means of an undo-log.

Each persistent object has its own metadata in a native HEAPO namespace. By analogy, persistent objects in HEAPO are similar to files in a file system, from a namespace perspective. In order to be used, an existing object must be attached. HEAPO's API and system organization is depicted in Figure 3.2.

HEAPO is designed to contain large size persistent objects, for example, in-memory key-value stores, XML-based documents, etc. It is not fit for maintaining small size kernel objects, for example, inodes, sockets, file objects. A persistent object is initially allocated in a 4KByte page. Expanding a persistent object involves attaching one or more pages from the persistent heap to the virtual address space of the caller.

Code Listing 3.6 exemplifies a simple program using HEAPO. This code creates a persistent linked list. The application creates a persistent object named "list" in line 12. If this object already exists (line 13), the application attaches it to its virtual address space. Line 15 obtains the starting address of the persistent object. If "list" is new, HEAPO reserves a fraction of the persistent heap

Listing 3.6 – HEAPO example code

```
1  #include <heapo.h>
2
3  struct list {
4    int data;
5      struct list *next;
6  };
7
8  int main(void) {
9    struct list *head, *node;
10     int i;
11
12     if (heapo_create("list") == 0)
13       heapo_map("list");
14
15     head = heapo_get_prime_node("list");
16     node = (struct list *) heapo_malloc("list", sizeof(struct list));
17     node->data = rand();
18     node->next = head;
19     heapo_set_prime_node("list", node);
20
21     heapo_unmap("list");
22     return 0;
23 }
```

for it. At this phase, the persistent page has not been allocated yet. The persistent node is allocated to the persistent object by calling `heapo_malloc()` (line 16). Once the application is done with accessing the persistent object, it detaches the persistent object from its name space (line 21).

3.4.5 NVML

The NVM Library (NVML) [27] is a set of open-source libraries specifically designed for persistent memory programming in C/C++. NVML is a user-space library providing constructs for developers to program access to byte-addressable, persistent memory while abstracting away much of the complexity of dealing with memory management, transactions, locking, lists, and other implementation challenges.

NVML is organized into a set of libraries, each one with a different purpose:

- libpmemobj — provides a transactional object store, providing memory allocation, transactions, and general facilities for persistent memory programming.

- libpmemblk — supports arrays of pmem-resident blocks of the same size that are atomically updated, e.g., a program keeping a cache of fixed-size objects.

- libpmemlog — provides a pmem-resident log file, useful for programs such as databases that append frequently to a log file.

- libpmem — provides low level persistent memory support. The libraries above are implemented using libpmem.
- libvmem — turns a pool of persistent memory into a volatile memory pool, similar to the system heap but kept separate and with its own malloc-style API.
- libvmmalloc — transparently converts all the dynamic memory allocations into persistent memory allocations. This allows the use of persistent memory as volatile memory without modifying the target application.
- libpmempool — support for off-line pool management and diagnostics.
- librpmem — low-level support for remote access to persistent memory utilizing RDMA-capable RNICs. The library can be used to replicate content of local persistent memory regions to persistent memory on a remote node over RDMA protocol.
- C++ bindings — aiming at providing an easier to use, less error prone implementation of libpmemobj.

Persistent Programming using NVML allow memory data structures to be the single representation for both manipulating and storing data. This avoids the need for translating different data formats, simplifying the code and improving execution performance.

Code listing 3.7 shows a very simple example of libpmemobj usage. Line 11 creates a PMEM object pool, which is a named memory region that can be used by NVML programs (it is implemented as a memory-mapped file on top of PM). The PMEM object pool has a *root object* that can be retrieved using NVML functions; in line 15, the `pmemobj_root()` functions returns the address of the root object as a PMEMoid, which is a relocatable address relative to the pool base. The `pmemobj_direct()` function called in line 16 converts the PMEMoid to a plain C pointer. The root struct contents are changed in lines 19—20, and the `pmemobj_persist()` function in line 21 ensures that the changes are committed to PM (e.g. calling libpmem lower-level functions to issue ordering barriers and flush volatile caches). The `pmemobj_memcpy_persist()` function in line 22 copies the contents of `buf` to `rootp->buf` and ensures this copy is committed to persistent memory. Finally, the pool `pmemobj_close()` function is called in line 25 to perform resources cleanup and close the pool.

Listing 3.7 – Example NVML Program with Atomic Transaction

```
1 #define POOL_NAME "mypool"
2 #define LAYOUT_NAME "mylayout"
3 #define MAX_BUF_LEN 10
4
5 struct my_root {
6   size_t len;
7   char buf[MAX_BUF_LEN];
8 };
```

```
9
10 int main(int argc, char *argv[]) {
11   /* create PMEM object pool */
12   PMEMobjpool *pop = pmemobj_create(POOL_NAME, LAYOUT_NAME, PMEMOBJ_MIN_POOL,
         0666);
13
14   /* get a direct pointer to the pool's root struct */
15   PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
16   struct my_root *rootp = pmemobj_direct(root);
17
18   /* populate the root struct */
19   char buf[MAX_BUF_LEN];
20   rootp->len = strlen(buf);
21   pmemobj_persist(pop, &rootp->len, sizeof(rootp->len));
22   pmemobj_memcpy_persist(pop, rootp->buf, buf, rootp->len);
23
24   /* close the PMEM object pool */
25   pmemobj_close(pop);
26 }
```

NVML also provides atomic transactions by means of an undo log in PM. In order to specify that a memory range will be part of a transaction, the `pmemobj_tx_add_range()` function must first be called prior to any modification. This function copies the original contents of the given memory region to an undo log, which can be used in case of transaction rollback.Also, NVML assumes that when you add the memory range you intend to write to it and the memory is automatically persisted when committing the transaction, so the programmer does not need to call `pmemobj_persist()` directly. Code listing 3.8 presents a program similar to the previous one, but with an atomic transaction, delimited by the NVML macros in lines 20 and 24. Before any memory contents are modified, they are added to the undo log in line 21.

Listing 3.8 – Example NVML Program

```
1 #define POOL_NAME "mypool"
2 #define LAYOUT_NAME "mylayout"
3 #define MAX_BUF_LEN 10
4
5 struct my_root {
6   size_t len;
7   char buf[MAX_BUF_LEN];
8 };
9
10 int main(int argc, char *argv[]) {
11   /* create PMEM object pool */
12   PMEMobjpool *pop = pmemobj_create(POOL_NAME, LAYOUT_NAME, PMEMOBJ_MIN_POOL,
         0666);
13
```

```
14   /* get a direct pointer to the pool's root struct */
15   PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
16   struct my_root *rootp = pmemobj_direct(root);
17
18   /* populate the root struct */
19   char buf[MAX_BUF_LEN];
20   TX_BEGIN(pop) {
21     pmemobj_tx_add_range(root, 0, sizeof(struct my_root));
22     rootp->len = strlen(buf);
23     memcpy(rootp->buf, buf, strlen(buf));
24   } TX_END
25
26   /* close the PMEM object pool */
27   pmemobj_close(pop);
28 }
```

Libpmemobj defines a set of macros that provides a static type enforcement, catching potential errors at compile time. For example, a compile-time error is generated when an attempt is made to assign a handle to an object of one type to the object handle variable of another type of object.

The most part of the pmemobj library functions are thread-safe. The library provides a pthread-like API for adding locks to PM data structures. There is no need to initialize those locks or to verify their state; when an application crashes, they are all automatically unlocked.

To improve reliability and eliminate single points of failure, data stored in an PMEM object pool can be automatically written to local or remote pool replicas, thereby providing a backup for a persistent memory pool by producing a mirrored pool set.

NVML provides many other capabilities, and is being actively developed and extended to other programming languages. The next sections present PyNVM and Persistent Collections for Java, and both use NVML for internal implementation.

3.4.6 PyNVM

PyNVM [68, 69] is a Python module written by Christian S. Perone providing Python bindings for the NVML libraries. These bindings use the Python CFFI package (C Foreign Function Interface for Python) and the NVML library itself.

Code Listing 3.9 provides an example of a simple PyNVM program. Lines 6—7 create and initialize a file using functions from the standard `os` and `fallocate` modules. Line 10 invokes the PyNVM module to obtain a persistent memory-mapped view of the file. Lines 12—19 exemplify how to write and read contents from the persistent memory-mapped file. Line 23 calls a PyNVM function to ensure that all modifications are flushed from volatile caches and committed to persistent media.

Listing 3.9 – Example PyNVM Program

```
1 import os
2 from nvm import pmem
```

```
from fallocate import posix_fallocate

# Open file to write and fallocate space
fhandle = open("dst.dat", "w+")
posix_fallocate(fhandle, 0, 4096)

# mmap it using pmem
reg = pmem.map(fhandle, 4096)

# Write on it and seek to position zero
reg.write("lol" * 10)
reg.write("aaaa")
reg.seek(0)

# Read what was written
print reg.read(10)
print reg.read(10)

# Persist the data into the persistent memory
# (flush and hardware drain)
pmem.persist(reg)
```

PyNVM also supports context managers providing syntactic sugar to avoid explicitly calling `pmem.persist` and `pmem.unmap`, as illustrated by Code Listing 3.10.

Listing 3.10 – Example PyNVM Program with Context Manager

```
import os
import numpy as np
from nvm import pmem
from fallocate import posix_fallocate

fhandle = open("dst.dat", "w+")
posix_fallocate(fhandle, 0, 4096)

# Will persist (pmem_persist) and unmap
# automatically
with pmem.map(fhandle, 4096) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")
```

PyNVM supports many NVML libraries, such as libpmem, libpmemlog, and libpmemblk, although it does not cover all the features from NVML.

3.4.7 Persistent Collections for Java (PCJ)

Persistent Collections for Java (PCJ) [28] is a library for Java objects stored in persistent memory recently published by Intel. The library aims to make programming with persistent objects feel natural to a Java developer, for example, by using familiar Java constructs when incorporating persistence elements such as data consistency and object lifetime. The implementation of PCJ uses the libpmemobj library from NVML.

PCJ provides 8 Java classes whose instances can persist (i.e. remain reachable) beyond the life of a Java VM instance:

- PersistentByteBuffer — similar to java.nio.ByteBuffer.
- PersistentTreeMap — a java.util.SortedMap whose keys and values are persistent objects.
- PersistentString — stores immutable string data.
- PersistentLong — a boxed long object.
- PersistentArray — an array of persistent objects.
- PersistentTuples — tuples of possibly mixed-type persistent objects.
- ObjectDirectory — a statically reachable map between String keys and persistent objects.
- MemoryRegionObject — a persistent object offering low level getters and setters for primitive integral types.
- Transaction — a construct for atomic grouping of operations on persistent data.

Code Listing 3.11 exemplifies a simple PCJ program. In the `doPut()` method, lines 6—7 create a SortedMap called `employees` in persistent memory and registers it in the ObjectDirectory. Lines 10—17 allocate PersistentByteBuffer objects from persistent memory and put them into the `employees` map. In the `doGet()` method in line 21, a reference to the `employees` maps is obtained from the ObjectDirectory, and lines 23—37 retrieve the employee data by their id and print the results in the console.

Listing 3.11 – Example PCJ Program

```
1 import lib.persistent.*;
2
3 public class PutGet {
4
5   public static void doPut() {
6     PersistentTreeMap<PersistentByteBuffer, PersistentByteBuffer> employees =
          new PersistentTreeMap<PersistentByteBuffer, PersistentByteBuffer>();
7     ObjectDirectory.put("employees", employees);
8
```

```
9       PersistentByteBuffer id = PersistentByteBuffer.allocate(8);
10      id.putLong(1457);
11      String name = "Bob Smith";
12      int len = name.length();
13      int salary = 25000;
14      PersistentByteBuffer data = PersistentByteBuffer.allocate(len + 4 + 4);
15      data.putInt(len).put(name.getBytes()).putInt(salary);
16
17      employees.put(id, data);
18    }
19
20    public static void doGet() {
21      PersistentTreeMap<PersistentByteBuffer, PersistentByteBuffer> employees =
            ObjectDirectory.get("employees", PersistentTreeMap.class);
22
23      long empId = 1457;
24      PersistentByteBuffer id = PersistentByteBuffer.allocate(8);
25      id.putLong(empId);
26
27      PersistentByteBuffer data = employees.get(id);
28
29      int len = data.rewind().getInt();
30      byte[] bytes = new byte[len];
31      data.get(bytes);
32      String name = new String(bytes);
33
34      int salary = data.getInt();
35
36      System.out.println("Employee id" + "\t" + "Name" + "\t" + "Salary");
37      System.out.println(empId + "\t" + name + "\t" + salary);
38    }
39  }
```

3.4.8 Managed Data Structures (MDS)

The Managed Data Structures (MDS) library [34], created by Hewlett-Packard Enterprise, provides a high-level programming model for persistent memory. It is designed to take advantage of large, random-access, non-volatile memory and highly-parallel processing. Application programmers use and persist their data directly in their application, in common data structures such as lists, maps and graphs, and the MDS library manages this data. The library supports multi-threaded, multi-process creation, use and sharing of managed data structures, via APIs in multiple programming languages, Java and C++.

A programmer creates a managed data structure using the MDS API. Code Listing 3.12 provides an example of a Java program that creates a ManagedArray of integers, populates the array, and

Listing 3.12 – MDS array example

```
1 ManagedIntArray inventory_i = ManagedInt.TYPE.createArray(size);
2
3 inventory_i.set(0,0);
4 inventory_i.set(1,1);
5 inventory_i.set(2,2);
6
7 int total = 0;
8 for (ManagedInt i: inventory_i) {
9     System.out.println("inventory: i: " + i.asInt());
10    total += i.asInt();
11    System.out.println("total so far: " + total);
12 }
13 System.out.println("Total inventory: " + total);
```

iterate over the elements.

When a programmer creates a managed data structure, like this ManagedArray, MDS allocates this object directly in the MDS Managed Space. This Managed Space is one large virtual pool of memory; a shared, persistent heap in which MDS allocates and manages MDS application objects, with the help of the Multi-Process Garbage Collector. MDS objects are allocated in the MDS Managed Space using create method calls on an object type; in contrast with a programmer calling the new method to create a standard Java object in the Java single-process volatile heap.

All MDS objects are strongly typed. Managed types support a createArray() method to create a managed array of that type; for example, ManagedInt.TYPE (the managed type for integers) has a createArray() method call to create instances of ManagedIntArray, a type for managed arrays of integers. MDS is designed to support primitive types, strings, data structures and records, and it enables users to specify their own record types composed of these MDS supported types. To avoid the user having to write a lot of boiler-plate code for a ManagedRecord type, MDS provides support for high-level specification of a ManagedRecord type using Java annotations.

To make an MDS object accessible beyond the process that creates it, object must be bound to a name in the MDS namespace, as exemplified in Code Listing 3.13 line 2. Another application can then look up the same MDS object by name (line 5), either concurrently or long after the original creator process has finished, and get back a reference to it in the MDS Managed Space.

MDS also allows isolating updates to managed objects using a transactional IsolationContext (Code Listing 3.13, lines 7—9), preventing potential conflicts with other updates being made concurrently by another thread or process.

3.4.9 Apache Mnemonic

Apache Mnemonic [5] is a Java-based non-volatile memory library for in-place structured data processing and computing. It provides unified interfaces for data manipulation on heterogeneous block and byte-addressable devices, such as DRAM, persistent memory, NVMe, SSD, and cloud network devices.

Listing 3.13 – MDS example of namespace binding and isolated execution

```
// Application #1
inventory_i.bindName("inventory_i");

// Application #2
ManagedIntArray inventory_i = ManagedIntArray.TYPE.lookupName("inventory_i");

isolated(() -> {
    inventory_i.set(3,3);
});
```

Mnemonic defines Non-Volatile Java objects that store data fields in persistent memory and storage. During the program runtime, only methods and volatile fields are instantiated in Java heap, Non-Volatile data fields are directly accessed via GET/SET operation to and from persistent memory and storage. Mnemonic avoids serialization/de-serialization of objects, reducing data movement and the amount of garbage produced in the JVM heap.

According to its proposal [6], Apache Mnemonic offers the following advantages:

- Provides an abstract level of viewpoint to utilize heterogeneous block/byte-addressable device as a whole (e.g., DRAM, persistent memory, NVMe, SSD, HDD, cloud network storage).
- Provides seamless support object oriented design and programming without adding burden to transfer object data to different form.
- Avoids the object data serialization/de-serialization for data retrieval, caching and storage.
- Reduces the consumption of on-heap memory and in turn to reduce and stabilize Java Garbage Collection (GC) pauses for latency sensitive applications.
- Overcomes current limitations of Java GC to manage much larger memory resources for massive dataset processing and computing.
- Supports the migration data usage model from traditional NVMe/SSD/HDD to non-volatile memory with ease.
- Uses lazy loading mechanism to avoid unnecessary memory consumption if some data does not need to use for computing immediately.
- Bypasses JNI call for the interaction between Java runtime application and its native code.
- Provides an allocation aware auto-reclaim mechanism to prevent external memory resource leaking.

Code Listing 3.14 shows an example of how to define a non-volatile class using Apache Mnemonic. Such classes must be abstract, implement the Durable interface and be marked with a `@DurableEntity`

Listing 3.14 – Apache Mnemonic - defining a non-volatile class

```
1 @DurableEntity
2 public abstract class Person<E> implements Durable, Comparable<Person<E>> {
3         E element; // Generic Type
4
5         @Override
6         public void initializeAfterCreate() {
7                 System.out.println("Initializing After Created");
8         }
9
10        @Override
11        public void initializeAfterRestore() {
12                System.out.println("Initializing After Restored");
13        }
14
15        @DurableGetter(Id = 1L)
16        abstract public String getName() throws RetrieveDurableEntityError;
17
18        @DurableSetter
19        abstract public void setName(String name, boolean destroy) throws
              OutOfPersistentMemory, RetrieveDurableEntityError;
20 }
```

annotation. Getters and setters also must be abstract, use special annotations and throw specific exceptions (OutOfPersistentMemory and RetrieveDurableEntityError). These classes also need to implement callback methods for initializing each object after creating and after restoring them.

Code listing 3.15 exemplifies how to create and retrieve durable objects using Apache Mnemonic. In line 2 we obtain a reference to a NonVolatileMemoryAllocator, the object that represents a named data store in persistent memory. Lines 5—7 exemplify how to create a durable object, set its attributes, and associate it with a key ID inside the allocator. Lines 11—17 show how to retrieve objects using the allocator object. Finally, line 20 shows how to close the allocator after use.

3.5 Conclusion

In this chapter, we have presented the concept of persistent memory, and recent works leveraging it to simplify persistence abstractions and improve performance. All these works have in common the fact that persistent data are represented or allocated in a different way than volatile data. Mnemosyne, Atlas, HEAPO, NVML, PyNVM and Apache Mnemonic require special memory allocators to be used for persistent data. NV-heaps, PCJ, MDS, and again Apache Mnemonic use specialized types for persistent objects. In addition, NV-heaps and NVML also use special pointers to refer to persistent objects.

Having a distinction between short- and long-lived data representations yields an *impedance mismatch* that adds complexity to software development. The three principles of orthogonal persistence described in Section 2.1 try to address this problem by supporting a uniform treatment of objects irrespective of their types, allowing values of all types to have whatever longevity is required. By using

Listing 3.15 – Apache Mnemonic - creating and using a non-volatile object

```
// create an allocator instance
NonVolatileMemAllocator act = new NonVolatileMemAllocator(1024 * 1024 * 8,
    "./pobj_person.dat", true);

// create a new non-volatile person object from this specific allocator
person = PersonFactory.create(act);
person.setName(String.format("Name: [%s]", "John Doe"), true);
// keep this person on non-volatile handler store
act.setHandler(keyidx, person.getHandler());

// fetch handler store capacity from this non-volatile storage managed by
    this allocator
long KEYCAPACITY = act.handlerCapacity();

// iterate non-volatile handlers from handler store of this specific
    allocator
for (long i = 0; i < KEYCAPACITY; ++i) {
  // restore person objects from this specific allocator
  Person<Integer> person = PersonFactory.restore(act, val, true);
}

// close allocator after use
act.close();
```

special types, allocators or pointers for persistent data, all the programming interfaces described above do not adhere to one or more of these principles.

In the next chapter we propose an orthogonally-persistent heap design specifically created for persistent memory.

4. A DESIGN OF ORTHOGONALLY-PERSISTENT HEAPS FOR PERSISTENT MEMORY

In Chapter 2, we have described how orthogonal persistence can be used to remove from the programmer the burden of explicitly handling persistence, but how its implementation on top of traditional storage adds complexity and decreases performance. In chapter 3, we described the current efforts to design programming language interfaces supporting persistent memory, but how the existing proposals still require the programmer to explicitly handle persistence (although in much more simplified way, since they do not require crossing the semantic boundary between memory data structures and files or databases).

In this chapter, we combine both approaches by leveraging persistent memory in an attempt to solve the performance and complexity issues historically associated with orthogonal persistence. We do this by introducing a design for the runtime environment for languages with automatic memory management, based on an original combination of orthogonal persistence, persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions. Such design can significantly increase programming and execution efficiency, as in-memory data structures are transparently persistent, without the need for programmatic persistence handling, and removing the need for crossing semantic boundaries.

The existence of different abstractions for representing data in memory and in persistent storage is not a fundamental necessity of programming languages, but an accidental result of the current technology that mandates the separation of fast, small, volatile main memory from slow, abundant, non-volatile secondary storage. If these two different layers are collapsed into persistent memory, then orthogonal persistence can overcome its past implementation challenges, and provide an interface that requires less code complexity and programmer effort.

4.1 Motivation

For about 30 years, between 1983 and 2001, an intense research agenda around orthogonal persistence was carried out. Many researchers invested effort creating programming interfaces and abstractions that could simplify software development and maintenance by removing the impedance mismatch between memory and storage data representations. However, implementing orthogonally-persistent systems on top of traditional storage posed performance challenges that prevented many of these abstractions from reaching mainstream software development.

But now, the imminent advent of persistent memory opens new possibilities for implementing orthogonally-persistent systems using technologies that are more naturally suited to the task. The main motivation for our design is to combine the previous studies on orthogonal persistence with modern research on persistent memory in order to enable the creation of software applications that are easy to both write and maintain, and have significantly superior performance than today.

4.2 System Assumptions

We assume the hardware platform to have the following characteristics:

- Non-volatile memory devices are accessed directly using CPU loads and stores;
- Volatile processor caches are still present;
- Non-volatile memory load/store latencies are similar to DRAM (an assumption also made by [20, 40, 44]).

From an operating system perspective, we assume it uses a PM-aware file system, which is designed specifically for persistent memory [77]. Such a file system provides direct access to PM, performing significantly better than disk- or flash-based file systems. The performance improvement comes from eliminating the block I/O layer, bypassing the page cache, and removing other steps rendered unnecessary by PM, as it can be directly addressed by the processor at byte granularity. Examples of such file systems are BPFS [23], PRAMFS [78], PMFS [33], Ext4/DAX [84], SCMFS [86], Aerie [81], NOVA [87], NTFS [21] and M1FS [71].

4.3 Requirements

Our proposal is based on the following requirements:

- Simplified development and maintenance of persistence code, by adhering to the OP principles;
- Backward compatibility with legacy source and executable code whenever possible;
- Generalizable design that can be applied to Object-Oriented languages with automatic memory management (e.g. garbage collection).

4.4 Persistent Heap Adherent to Orthogonal Persistence Concepts

Orthogonal persistence (OP) (described in Chapter 2) provides abstractions to address the impedance mismatch between memory and storage by means of supporting a uniform treatment of objects irrespective of their types, allowing values of all types to have whatever longevity is required.

We propose that object-oriented languages with automatic memory management can implement OP concepts by means of a *persistent heap* that can be reused across executions. In this work, we consider 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.

In order to determine the longevity of objects, we advocate the use of *persistence by reachability*, i.e. all objects still reachable from *roots of persistence* are considered persistent, while objects that are unreachable can be disposed. We also advocate using as roots of persistence the static variables

of all classes that are persistent and the stack of all active threads. This approach leverages the same reachability criteria used for Java garbage collection [39], and has been successfully used in previous OP Java implementations [9, 51, 57].

In the next sections we list what needs to be stored in the persistent heap in order to support persistence by reachability of both class attributes and stack references.

We use Java as the programming language for our examples, but the design is generalizable to other object-oriented languages with automatic memory management.

4.5 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).

Using persistence by class attribute reachability, 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.

Code Listing 4.1 is an example of persistence by class attribute reachability in Java.

Listing 4.1 – An example of persistence by class attribute reachability

```
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 }
```

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 4.1.

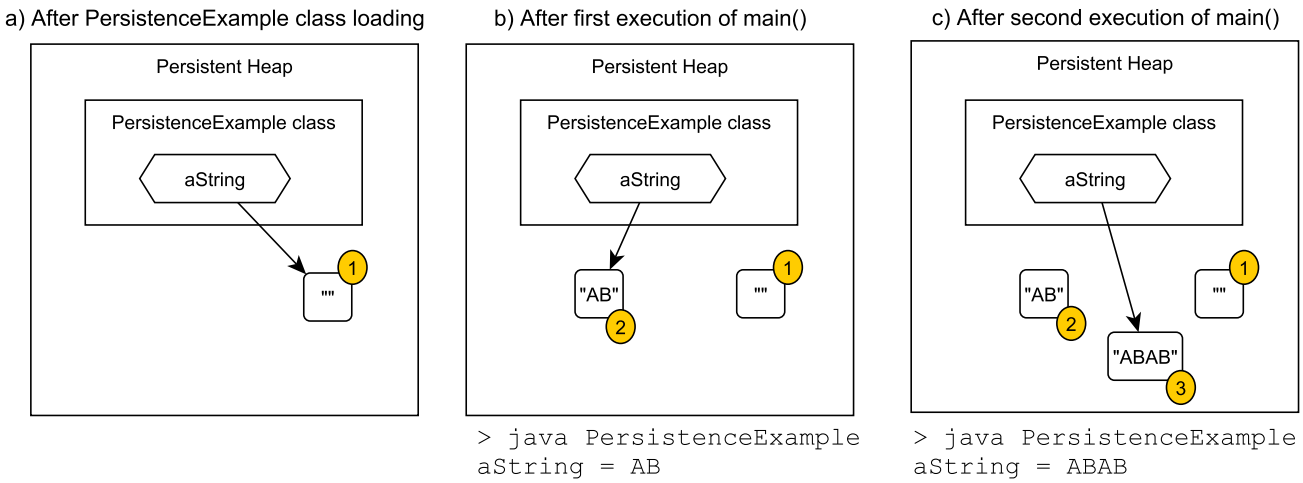


Figure 4.1 – Example of persistence by class attribute.

1. When the `PersistenceExample` class is loaded into the persistent heap (at an arbitrary location), its class attribute `aString` points to the String object (1), which contains the value "".

2. After the first execution of the `main()` method, a new String object (2) containing the value "AB" is created, and the `aString` attribute is now referencing it. The String object (1) is available for disposal, since no class attribute references it.

3. After the second execution of the `main()` method, a new String object (3) containing the value "ABAB" is created, and the `aString` attribute is now referencing it. The String object (2) also becomes available for disposal, since no class attribute references it anymore.

In the previous example, objects that were directly referenced by class attributes were considered persistent. It is also true for objects that are indirectly referenced by class attributes as well. Code Listing 4.2 illustrates this case.

Listing 4.2 – An example of persistence by indirect class attribute reachability

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

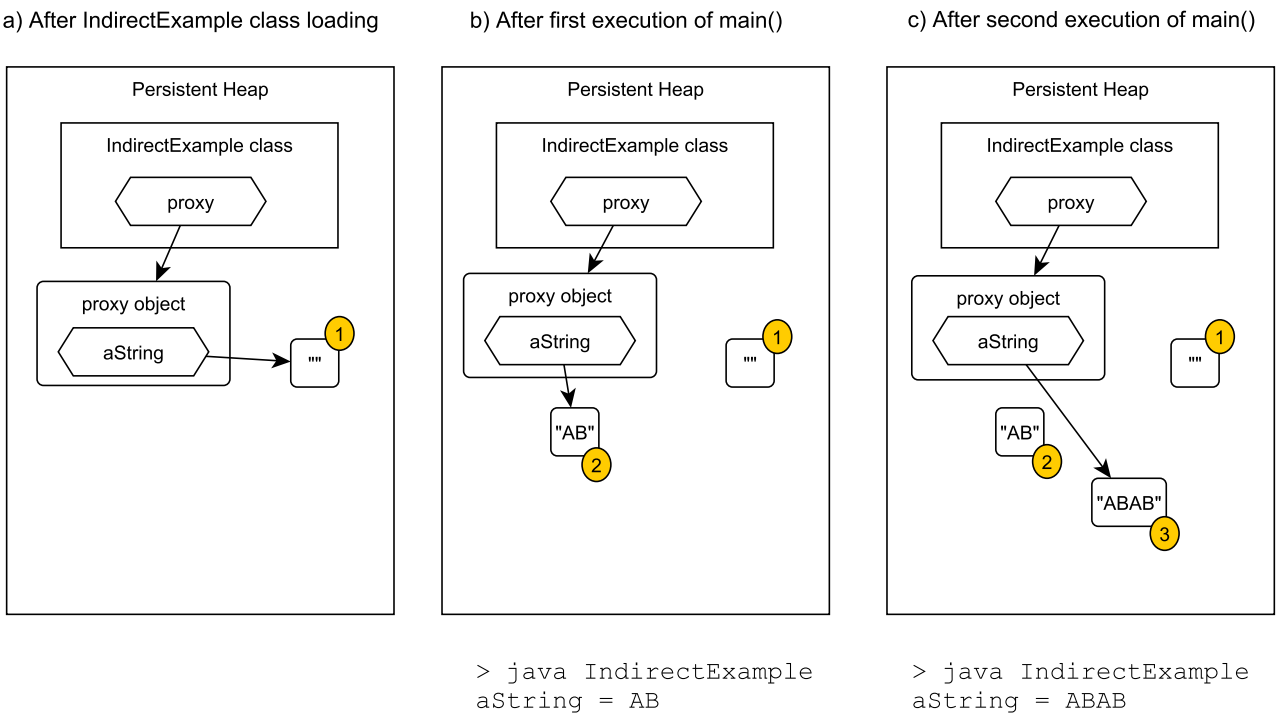


Figure 4.2 – Example of persistence by indirect class attribute reference.

```
13     }
14 }
```

Proxy is a simple class which contains one instance attribute called `aString`. The 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). As Java Strings are immutable, every concatenation generates a new String object in the heap. It is similar to the previous example, but now the references from a class attribute are indirect.

4.6 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.

Using persistence by stack reachability, 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.

Code Listing 4.3 is an example of persistence by stack reachability in Java.

Listing 4.3 – An example of persistence by stack reachability

```
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 }
```

Line 12 creates a new StackExample object in the heap. The stack frame of the `main()` method contains a variable `ex` referencing an 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 4.3.

Persistence by stack reachability allows the complete execution state of a program to be automatically persisted, 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.

4.7 Storage and Management of Persistent Heaps

As previously stated, we consider the term *persistent heap* as referring to the persistent state that can be bound to a program, which may include the heap itself and other elements of the persistent

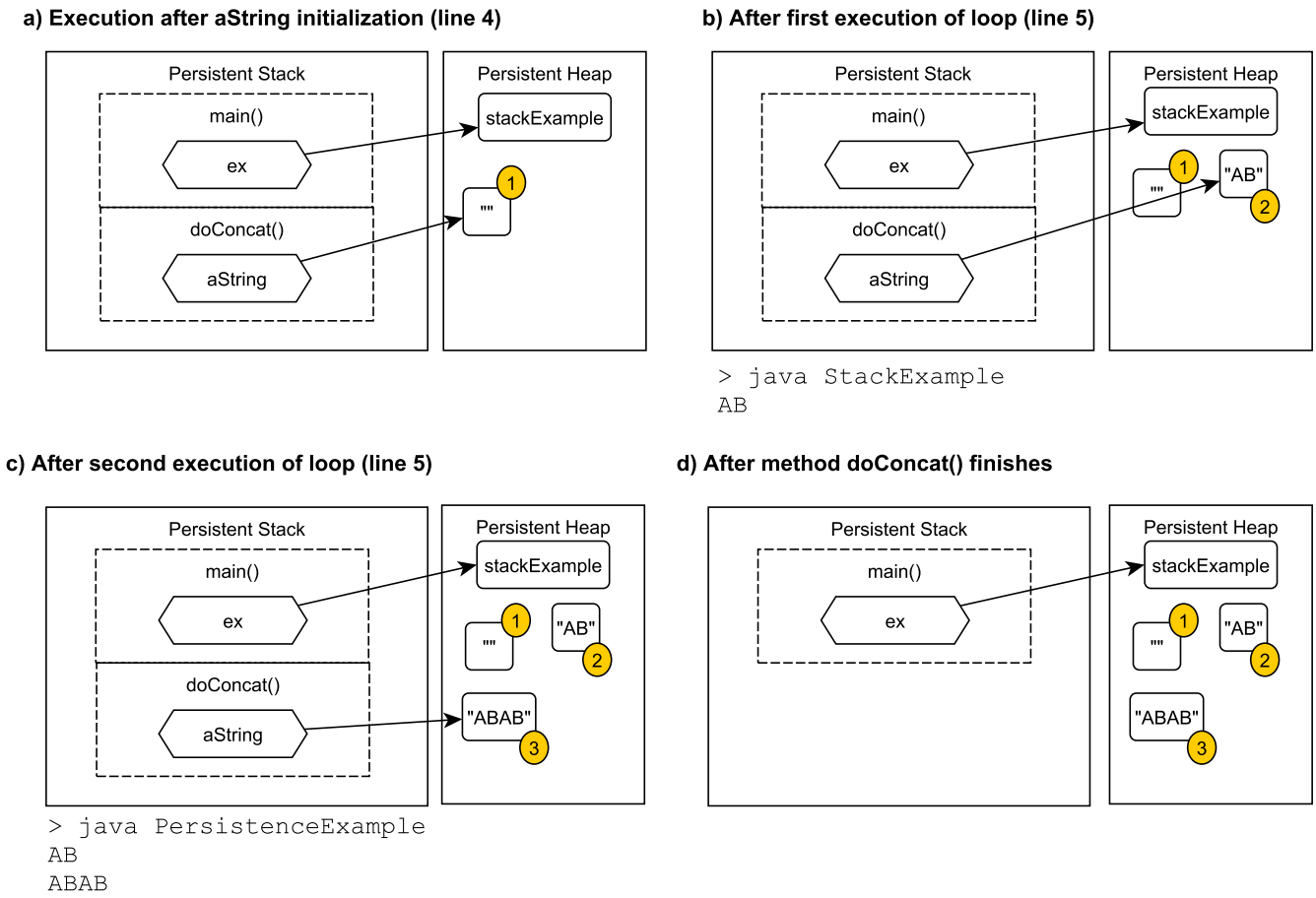


Figure 4.3 – Example of persistence by stack reference.

state, such as threads, stacks, loaded types (classes), JVM metadata, and other data as required by the implementation.

We assume a PM-aware file system providing namespace management and user access control to chunks of persistent memory [77]. The file system contains persistent heap files, which can be mapped to the addressing space of any process via *persistent memory programming*, either by directly calling the `mmap()` system call or using higher-level APIs for PM such as [20, 22, 27, 40, 44, 82]. Once mapped to the addressing space, the process can manipulate persistent memory directly, without page caches. This scenario is depicted on Figure 4.4

When the runtime environment of a program is invoked (e.g. JVM), the file containing the persistent heap can be provided (for example, as an argument).

In this work, we explore 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 to a single process, as well as multiple processes to a single persistent heap (i.e., 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.

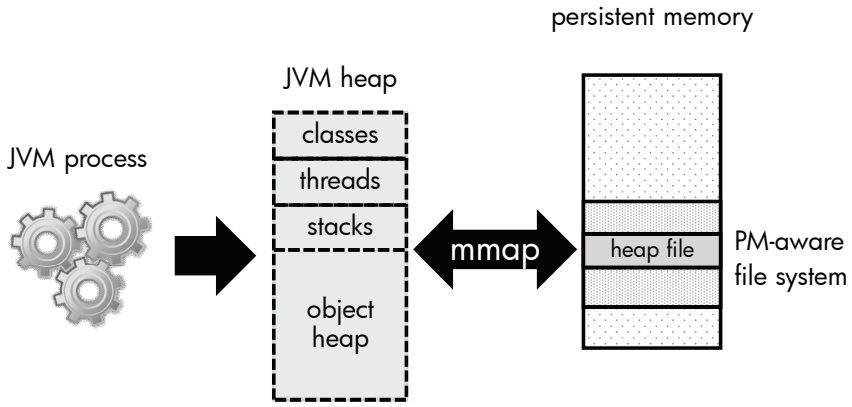


Figure 4.4 – persistent memory hosting an in-memory file system, which contains a persistent heap file

4.7.1 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 4.5) is available to be bound to a new process, which can be either from the same program or a different one.

Code Listing 4.4 provides an example of a program using data persistence. The program is always executed from the beginning in every invocation. Line 32 verifies if the persistent object `addrListByName` is uninitialized, which will be true in the first execution. It will be initialized in subsequent executions using the same persistent heap.

Another example of data persistence is the original implementation of orthogonal persistence described in [11].

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 Code Listing 4.4 as an example, and assuming a JVM with support for orthogonal persistence, it could be called by the command line `java -persistent-heap=heapfile.ph AddressList`. In that example, the program would be bound to the persistent data state contained in `heapfile.ph`.

Listing 4.4 – An example of persistence using data state binding

```
1 /**
2  * Class representing a single address.
3  */
4 public class Address {
5   private String name;
6   private String phoneNumber;
7   public Address(String name, String phoneNumber) {
8     this.name = name;
9     this.phoneNumber = phoneNumber;
10   }
```

```
11   public String getName() {
12     return this.name;
13   }
14   public String getPhoneNumber() {
15     return this.phoneNumber;
16   }
17 }
18
19
20 import java.util.Map;
21 import java.util.HashMap;
22
23 /**
24  * Program to maintain an address list
25  * using Orthogonal Persistence (data state binding).
26  */
27 public class AddressList {
28
29   private static Map<String,Address> addrListByName;
30
31   public static void main(String[] args) {
32     if (addrListByName == null) {
33       // initialize address list
34       addrListByName = new HashMap<String,Address>();
35     }
36     if (args.length >= 2) {
37       // add a new entry
38       addrListByName.put(args[0], new Address(args[0], args[1]));
39     }
40     // list existing entries
41     for (Map.Entry<String, Address> entry : addrListByName.entrySet()) {
42       Address addr = (Address) entry.getValue();
43       System.out.println("Name: " + addr.getName());
44       System.out.println("Phone Number: " + addr.getPhoneNumber());
45     }
46   }
47
48 }
```

4.7.2 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 4.5 and 4.6) 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.

Smalltalk virtual images [38] are a good example of execution persistence. Existing Smalltalk Virtual Machine (VM) implementations use files to store the virtual image. A Smalltalk VM mapping its virtual image to persistent main memory would be an instantiation of an in-memory persistent heap as we define it.

Merpati [80] is a prototype implementation of checkpoints of the JVM execution state in Java. It proposes a mechanism similar to the one we described, with two main differences: a) it is implemented using traditional file storage; b) the JVM execution is suspended at arbitrary moments for checkpointing, while in our proposal the persistent heap is manipulated directly during program execution without interruptions or checkpoints.

4.8 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. In order to accomplish this, it is necessary to make modifications to the persistent heap in the context of *failure-atomic transactions*. Persistent memory programming APIs such as [27,82] provide failure-atomic transactions by means of transactional memory and journaling, respectively.

It is also desirable to provide ways for the programmer 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*), since they can be easily expressed by synchronized methods and/or blocks. In the context of Java, `synchronized` methods and blocks define locks around specific objects by means of calls to `monitorenter` and `monitorexit` opcodes. Atlas [20] introduces a similar approach for the C language, and [83] demonstrates that Java synchronization locks can be used to express transactional semantics.

4.9 Challenges

There are many challenges and open points that must be addressed in order to have a robust persistent heap solution. In the next subsections, we list some of these challenges. Many of them are not specific to OP, being applicable to other memory persistence approaches as well. In this work we have addressed some of these points, as described in Section 5.1, while others must be addressed by future research works.

4.9.1 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. The most commonly proposed solution for this problem is to provide a Persistent Object Store (POS) [10] holding persistent objects that can be accessed by programs.

There are many design alternatives for a POS:

- POS as a library of a specific programming language: the POS is implemented by a library written in a specific programming language, and only programs written in that language can access data. A recent example is provided in [3].

- POS as part of the runtime system of a specific programming language: the execution environment of a specific programming language is modified to provide a POS. Only programs of that language can access data. Our proposal fits into that category, and other examples are provided by [51], [80] and [16].

- POS integrated by design into a specific programming language: the programming language is designed with orthogonal persistence support as a requirement. Only programs of that language have access to persistent data. One example is PS-algol [11].

- POS as a separate store supporting multiple programming languages: the POS is a separate process, providing support to multiple separate programs and languages, which may be in a distributed environment. This is the approach followed by most Object-Oriented Database Management Systems (OODMBS) [7]. Some examples are [36] and [62].

4.9.2 Type Evolution

Each object in a heap belongs to a specific type. In our proposed design, the type definition is contained by the persistent heap (unless it is defined by the programming language, such as primitive types in Java [39]). In this scenario, 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.

Farkas and Dearle [35] described system evolution as encompassing evolving data, evolving code, and evolving types. They propose a mechanism for type evolution based on a persistent store browser, i.e., a tool that traverses object graphs applying a function to the objects it encounters. The browser could then find and update all the instances of a given type in a persistent store. However, they identified some limitations:

1. The impossibility to reach certain values due to encapsulation.

2. Evolved code needs to be bound to evolved data, and evolved data needs to be bound to evolved code, creating a circular dependency.

3. The lack of support for type evolution.

In order to address these limitations, they propose a reflection mechanism called Octopus, which allows all bindings in an arbitrary graph of objects to be examined and manipulated.

Atkinson and Jordan [8] describe two approaches for class evolution. The first one is to handle evolution by external action, from outside the programming environment. The PJama prototype (described in Chapter 2) implemented an evolution tool that operates off-line in a quiescent store. One of the limitations of this approach is that such tools are specific for a JVM implementation. The second approach is to provide an evolution mechanism as part of the programming model, for example, by extending the reflection API to modify classes at runtime, an approach used by Smalltalk. The Octopus mechanism would also fit in that category.

A recent paper by Piccioni et al. [70] proposes a mechanism, called ESCHER, which automatically updates objects from one class version to another using information provided by abstract syntax trees, that can be augmented by information stored in Integrated Development Environments (IDEs) during program development.

4.9.3 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. It also must provide ways for the programmer to execute concurrent code without undesirable effects.

Blackburn and Zigman [15] discuss the challenges that OP faces handling concurrency and consistency, and group the existing solutions in two approaches: transactions and checkpoints.

Transactions

In databases, the main constructs to handle these concerns are transactions. Transactions are a group of operations that exhibit Atomicity, Consistency, Isolation and Durability (ACID) properties. According to the authors, OP would require that computation should take place only from within ACID transactional contexts, in order to satisfy the principle of persistence independence. However, since a transaction invoked from within a transaction becomes a nested transaction that can only be successfully committed along with the parent one, the whole program execution would become a single transaction. Such a coarse-grained transaction scope limits its usefulness for controlling concurrency within a program. In other words, the transaction approach only makes sense when transactions can be invoked from non-transactional context. For OP implementations persisting execution state (as described in section 4.7.2) this would be a challenge, but transactions would be more feasible for implementations that persist only data (see section 4.7.1).

Checkpoints

In addition to the inherent challenges of integrating the concepts of OP and ACID transactions, there are also obstacles for implementing transactions as a mechanism naturally integrated to specific programming languages, such as Java [8]. An alternative approach described by the authors is for

the virtual machine to periodically take snapshots, or checkpoints of its internal state, and resume from these checkpoints in case of unexpected termination. The challenge of this approach is how to identify globally stable and consistent states in complex multi-threaded programs in order to capture them in checkpoints.

PEVM [51] addresses consistency by means of checkpoints. There is a persistent store in disk and an object cache (heap) in memory. When an object is written to the persistent store and removed from the object cache, its pointer is replaced by the location of the object in the persistent store (a process known as *pointer swizzling*). Objects are written to the persistent store during checkpoints. The first step for a checkpoint is calling *checkpoint listeners* that can be defined by the programmer, followed by stopping all threads but the one making the checkpoint. It then does a first pass to identify objects that needs to be persisted, and a second pass to actually save the objects in the persistent store. PEVM persists only data, and considers the execution state is volatile.

Other examples of checkpointing in Java are described in [41], [16], and [80].

Lumberjack [43] is a log-structured persistent object store that is used to support the Persistent Abstract Machine layer of the Napier88 system, although it was designed to be generic enough to support other systems. It can support both concepts of transactions and checkpoints, and applies the log-structured approach, consisting of a circular log for holding both data and meta-data. Its benefits are efficient commits and support for incremental parallel garbage collection.

Consistency and transactional properties are also important when dealing with concurrency. Berryhill et al. [13] describe potential issues and alternative solutions to manage concurrent access to a shared heap in persistent memory.

4.9.4 Persistent Bugs

A system with Execution Persistence may incur in persistent deterministic bugs that crash or impair subsequent executions.

One alternative for dealing with persistent bugs would be resetting the system to its initial state, which may not be acceptable for many applications. Another alternative would be to directly modify the persistent heap to get rid of the bug, which may require a special debugging program.

4.9.5 External State

Some objects and variables represent state that is external to the virtual machine. Notable examples are networking resources (e.g. sockets), files, and elements that vary from one system to another, such as locale.

Atkinson and Jordan [8] describe the main challenges that OP faces handling external state, in the context of the PJama implementation. They highlight that 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 re-initialized on each run. One solution approach is to somehow identify variables representing external state, treat them as a cache that is discarded when the program

is finished (either due to graceful termination or failure) and re-establish the active state during restart, which may require saving metadata about the state to be re-established. For example, a file descriptor `FD` based on a file named `FN` would not be valid across restarts, but can be reopened returning an equivalent descriptor `FDnew` based on `FN`. Still, the program must ensure that FDnew is still consistent with program semantics. This is even more challenging when the whole execution state is persisted (as described previously in section 4.7.2. Networking communication is particularly problematic.

These problems cannot be solved in a way that is completely transparent to the programmer, and are difficult to solve without creating specific libraries/frameworks.

One example is how external state is handled by PEVM [51]. The special static method `OPTransient.mark()` is used to mark objects as transient (this is required since the transient keyword of the Java language cannot be used since its definition conflicts with OP's definition of transient state). These objects can implement the `OPResumeListener` interface to automatically re-establish object state during virtual machine restart, as well as the `OPResumable` interface that is invoked before the object is first used to perform lazy initialization activities.

The Orthogonal Persistence for Java (OPJ) specification [46] proposes to add the `transient storage` modifier to the language as a means to identify variables that must be reset to a default state upon resumption. Classes can register interest in runtime life-cycle events, such as checkpoint, resume and shutdown, and take appropriate action.

4.9.6 Security

When using a persistent heap, potentially sensitive data becomes vulnerable to unauthorized access. To circumvent that, persistent heap data should be encrypted. Software-based encryption is possible but would have a huge performance impact. Hardware-based encryption provided by the underlying platform would probably be the most efficient mechanism.

Security of persistent memory systems is an important and broad research topic, and is beyond the proposed scope for this work.

4.10 Conclusion

In this chapter we have presented a design for a persistent heap based on an original combination of orthogonal persistence, persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions.

In the next chapter we present JaphaVM, a prototype to validate and evaluate the effectiveness of such design.

5. JAPHAVM: AN ORTHOGONALLY-PERSISTENT JAVA VIRTUAL MACHINE DESIGNED FOR PERSISTENT MEMORY

In this chapter we present JaphaVM, a prototype to validate and evaluate the effectiveness of the concepts proposed in Chapter 4. We start by describing the main design decisions behind JaphaVM, and then proceed to present and analyze experimental results.

5.1 JaphaVM Design

5.1.1 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 garbage collection 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 [57], followed the approach of bytecode transformation, allowing portability across different Java Virtual Machines (JVMs). Other implementations, such as PJama [9] and PEVM [51], 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, in order to have complete control over the data structures used to manage heap, stacks, threads, and type definitions. Since the mentioned OP-oriented JVMs are not available anymore, only current JVMs can be used as implementation baseline. Our choice was JamVM [55], due to its simple and straightforward implementation.

5.1.2 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, 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 [37] V0.92 open source implementation was used. Some modifications to GNU Classpath were made to address design issues, as will be described in Section 5.1.5.

5.1.3 Initial Design and its Limitations

The baseline JamVM version uses an anonymous, non-file-backed memory-mapped area via the `mmap` function in order 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. Functional tests performed on this initial implementation answered affirmatively research question 1, i.e., that "*we can use a persistent heap on top of persistent memory to transparently store objects across multiple program executions*". No specific memory management features for persistent objects were implemented, using the standard garbage collection mechanisms, and the results of functional testing addressed research question 2, demonstrating that "*it is possible to leverage automatic memory management mechanisms to determine the persistence lifecycle of ordinary objects*".

However, consistency was not guaranteed in the presence of failures, as confirmed by tests where the prototype process was randomly killed.

5.1.4 Final Design with Transactional Support

Our current persistent heap implementation uses the NVML pmemobj library [27] for manipulating persistent heap data in order to ensure fault-tolerance, as NVML provides journaled transactions via an undo log in PM. As consequence, the persistent heap is stored in an NVML *memory pool*, which is mapped to a file in PM. NVML has been chosen for currently being one of the most active and widely adopted persistent memory programming API, and providing a suitable transactional model.

We have employed failure-atomic transactions provided by NVML, as described in more detail in the next sections. The final prototype implementation, using this mechanism, successfully addressed research question 3, demonstrating a way "*to automatically ensure the consistency of a persistent heap in the presence of failures*".

5.1.5 Summary of JVM Modifications

The next sections summarize the main 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 explained later. The full source code can be found at [65], and Appendix A contains a more detailed description of all modifications made to create JaphaVM.

Invocation Arguments

In order to allow the invocation of the JamVM with a persistent heap, we have added a new invocation argument `-persistentheap:<heapname>` to alternate between persistent and non-persistent modes.

Persistent Heap

As mentioned in Section 5.1.3, the anonymous memory mapping used in the first JaphaVM version was replaced by an NVML memory pool in the final version. We created a struct called `PHeap` (Code Listing 5.1) that contains heap data and metadata, and acts as the memory pool's root data structure. When the JVM is executed with a persistent heap for the first time, the memory pool is created using the `pmemobj_create()` function, and opened in posterior executions with `pmemobj_open()`.

Listing 5.1 – PHeap struct

```
1 typedef struct pheap {
2   void *base_address;
3   Chunk *freelist;
4   Chunk **chunkpp;
5   unsigned long heapfree;
6   unsigned long maxHeap;
7   char *heapbase;
8   char *heapmax;
9   char *heaplimit;
10  nvmChunk *nvmfreelist;
11  nvmChunk **nvmChunkpp;
12  unsigned int nvmFreeSpace;
13  unsigned int nvmCurrentSize;
14  unsigned long nvm_limit;
15  OPC opc;
16  char* utf8_ht[UTF8_HT_SIZE];
17  char* bootCl_ht[BOOTCL_HT_SIZE];
18  char* bootPck_ht[BOOTPCK_HT_SIZE];
19  char* string_ht[STRING_HT_SIZE];
20  char* classes_ht[CLASSES_HT_SIZE];
21  char* monitor_ht[MONITOR_HT_SIZE];
22  char* zip_ht[ZIP_HT_SIZE];
23  char nvm[NVM_INIT_SIZE];
24  char heapMem[HEAP_SIZE];// heap contents
25 } PHeap;
```

`PHeap` member `heapMem` provisions the actual heap allocation space. Members `heapFree`, `maxHeap`, `heapBase`, `heapMax`, `heaplimit`, `freelist` and `chunkpp` hold heap metadata. Members `nvmFreeList`, `nvmChunkpp`, `nvmFreeSpace`, `nvmCurrentSize`, and `nvmLimit` contain pointers for a separate chunk area specific for non-heap metadata, provisioned in member `nvm`. Struct member `opc` contains a pointer to an `OPC` data structure containing additional metadata used for managing internal VM hash tables and Garbage Collection, among others.

Type Definitions and Compiled Methods

We need to ensure that all data structures that are referenced by heap objects, such as type definitions and compiled methods, are persisted and can be retrieved in future executions. Many of these structures were stored outside of the heap allocation space in the baseline JamVM implementation, using a function called `sysMalloc()`.

We have created a new function called `sysMalloc_persistent()` (and corresponding `sysFree()` and `sysRealloc()` functions) storing data in the non-heap metadata area, managed using a chunk-based approach similar to the one used by the JamVM heap.

Internal Hash Tables

JamVM uses a set of internal hash tables to keep track of loaded symbols, classes, and other data that must be stored across JVM executions. This was accomplished by modifying the `gcMemMalloc()` function implementation to point it to `PHeap` members ending with `_ht` containing the tables.

Failure-atomic Transactions

In order to meet the consistency requirements described in Section 4.8, 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`.

In order to enable programmers to define the transactional behavior of applications using `synchronized` methods and blocks, we have modified the `monitorenter` and `monitorexit` opcodes. 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 Garbage Collection (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: a) 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; b) 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 4.9.5, 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 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 [39] 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 [51] and PJama [9], exposing to developers an API for making classes and objects *resumable*, i.e., 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. In order 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 5.2). In order 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 re-initialize 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.

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 [31]. On the other hand, it requires small programmer involvement when compared to traditional file- or database-oriented approaches, so we deemed it a reasonable compromise for our prototype. A similar approach has been taken by the Apache Mnemonic library, described previously in Section 3.4.9.

Listing 5.2 – OPRuntime class definition

```
1 /**
2  * This class registers OPResumeListener objects for execution
3  * upon JVM re-initialization.
```

```
4  */
5 public class OPRuntime {
6
7   /**
8    * Adds a new listener.
9    */
10  public static void addListener(OPResumeListener listener);
11
12  /**
13   * Adds a new listener to a class that has a
14     * static resume() method.
15   */
16  public static void addStaticListener(Class<?> clazz);
17
18  /**
19   * Method called by the JVM runtime to re-initialize
20     * OPResumeListener objects.
21   */
22  public static void resumeAllListeners();
23
24 }
```

5.1.6 Prototype Limitations

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;
- 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 vs. execution persistence — the current version supports only data persistence (see Sections 4.7.1, 4.7.2);
- Type evolution — JaphaVM currently does not support type evolution.

Despite these limitations, the current JaphaVM prototype can execute a wide range of complex programs, as exemplified in the next sections, and functions as a vehicle for both performance and development complexity experiments and evaluation.

In this section we have described the key design decisions for implementing JaphaVM. A more in-depth description of JaphaVM implementation can be found in Appendix A. In the next section we evaluate JaphaVM's effectiveness and efficiency.

5.2 JaphaVM Evaluation

In order to assess the effectiveness of our design, and to address research question 4 "*what is the performance of an orthogonally-persistent system using a persistent heap relative to traditional storage approaches?*", we verified the JaphaVM 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 [1, 47].

In order to assess both development effort and execution performance, we have used two different workloads: a) the OO7 benchmark; and b) a modified version of the Lucene search engine. Both are described in the next sections.

5.2.1 Experimental Setting

Both OO7 and Lucene experiments were executed on a 32-core machine with 244GB of main memory running Ubuntu Linux 12.04. All experiments were executed three times to identify variation and outliers. The results of the multiple executions were generally consistent, and the average results are presented.

Since computers using NVM to implement persistent memory are not yet available, we used DRAM to simulate PM, an approach employed by previous studies [20, 44]. An area of 32GB was reserved for a PM-aware file system: M1FS [71], a configuration optimized for working directly with byte-addressable memory storage. JaphaVM was executed using a persistent heap with capacity of 8GB, without considering heap metadata.

5.2.2 OO7 Benchmark

The OO7 benchmark [19, 45] was originally aimed at object-oriented database management systems (OODBMS), and used in previous Java OP research [36, 51, 57].

This benchmark in a first step creates a database of objects combined in a tree-like hierarchy, 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. The size of the OO7 database is configurable.

We have chosen the OO7 benchmark for a number of reasons. Since it was used in previous Java OP works, it permits a basic level of comparison. It models both read- 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 5.2.3 we discuss a different application that uses files as the backing store.

Experiment Description

Tests with OO7 were executed in the following distinct configurations:

I **Disk-based DB** — backed by a traditional disk-based database (PostgreSQL version 9.1), using the Hibernate framework version 3.2 for ORM;

II **Memory-based DB** — backed by the H2 in-memory database (version 1.3.176), also using Hibernate;

III **JaphaVM** — backed by a modified version of the benchmark using JaphaVM's 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 persistent memory environment. JaphaVM was executed separately with three approaches for handling transactions (see Section 5.1.5):

1. Automatic Transactions, where each Java opcode modifying heap data is a self-contained transaction;
2. User Transactions, where the whole traversal is a single transaction within the bounds of a `synchronized` block;
3. No transactions, where there is no transactional overhead but also no fault tolerance (included for reference).

We ran each configuration with three different database size presets (payloads): tiny, small, and medium. These presets are described in detail in Table 5.1. In order 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.

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.

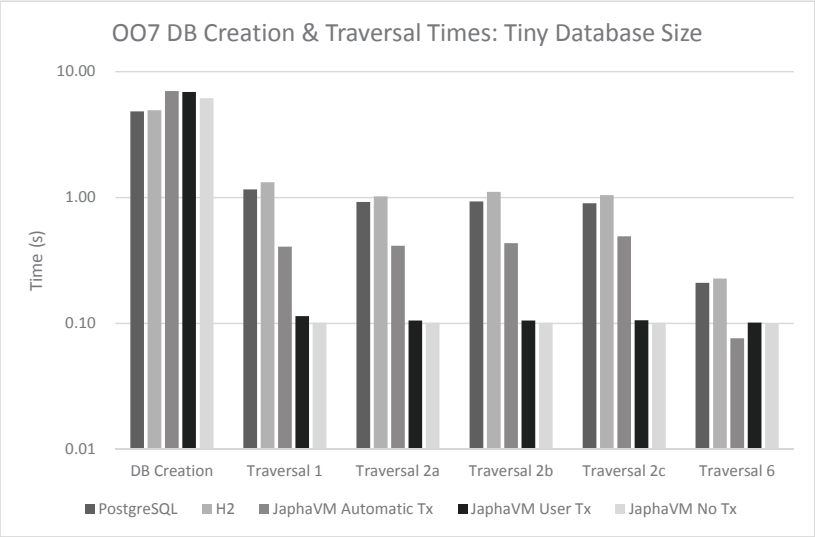
Table 5.1 – Properties of OO7 database size presets.

Property	Tiny DB	Small DB	Medium DB
# modules	1	1	1
# assemblies	40	1,093	1,093
# base assemblies	27	729	729
# composite parts	50	500	500
# base assemblies per complex assembly	3	3	3
# assembly levels	4	7	7
# complex assemblies	121	3,280	3,280
document size	2 MB	2 MB	20 MB
# atomic parts / composite parts	4	20	200
# atomic parts	200	10,000	100,000
# connections / atomic part	3	3	6
# connections	600	30,000	600,000
# updates T2a	54	2,184	2,183
# updates T2b	162	11,475	99,892
# updates T2c	486	39,348	393,019
manual size	10 MB	100 MB	1,000 MB
Total size in JaphaVM heap	60 MiB	160 MiB	710 MiB

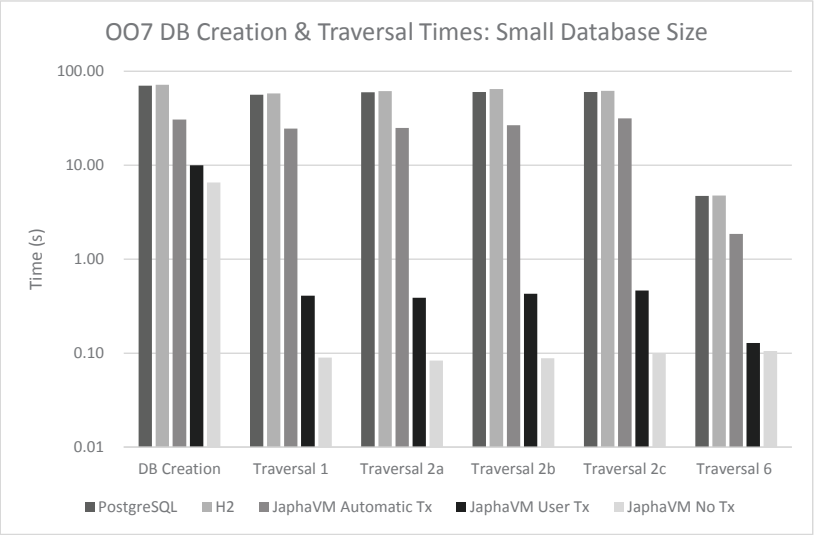
Figures 5.1a—5.1c compare the execution time of creating the databases and each traversal for the different database sizes. We can observe in Figures 5.2a—5.2e that all execution scenarios have similar trends, despite the execution times being orders of magnitude different. Figures 5.2a—5.2e show the same data with a different perspective, highlighting how the execution time increases for different database sizes in each traversal.

Tables 5.2—5.4 show I/O and execution counters for creating each database size, while Tables 5.5—5.7 show the same information for Traversal 1 on each database size. The lines "read characters" and "written characters" show the total number of characters read and written through system calls invoked by the program, irrespective whether the operation was satisfied from the page cache or from the backing storage. Console I/O is also included in the read/written characters count. The lines "read system calls" and "write system calls" show the number of system calls invocations. The remaining lines display the number of voluntary and involuntary context switches.

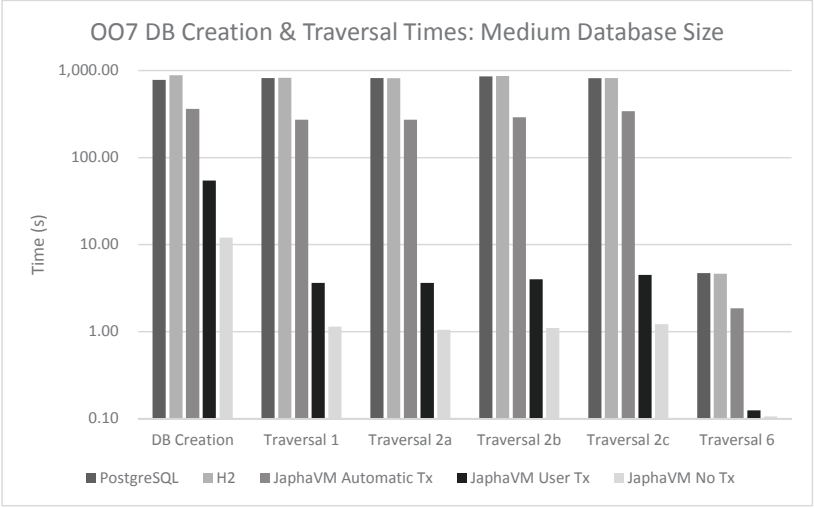
As seen on Figure 5.1a, for the tiny database size JaphaVM takes about two seconds longer than PostgreSQL and H2 to create the database, due to memory pool initialization. However, for the creation of the small and medium databases (Figures 5.1b—5.1c), JaphaVM scenarios execute one or two orders of magnitude 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 5.2—5.4 (we show a single column for JaphaVM because I/O counters for all transactional scenarios were identical).



(a) Comparison of execution times for tiny database size.

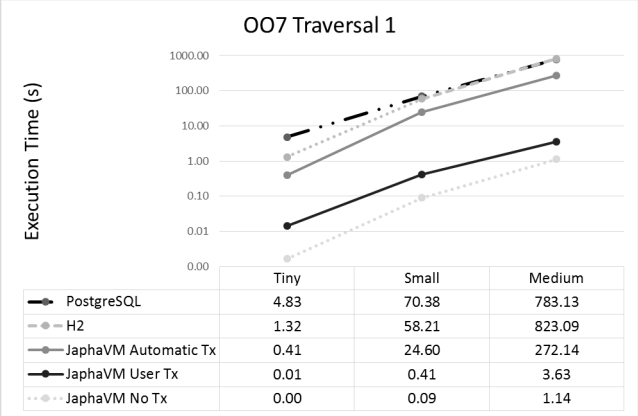


(b) Comparison of execution times for small database size.



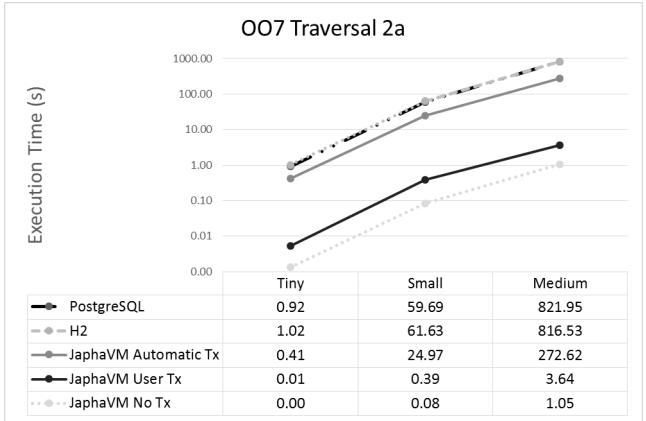
(c) Comparison of execution times medium database size.

Figure 5.1 – Comparison of execution times for OO7 traversals of different db sizes (secs, log scale)



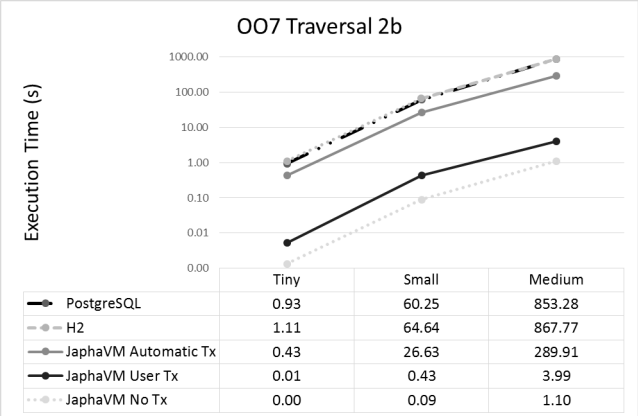
	Tiny	Small	Medium
PostgreSQL	4.83	70.38	783.13
H2	1.32	58.21	823.09
JaphaVM Automatic Tx	0.41	24.60	272.14
JaphaVM User Tx	0.01	0.41	3.63
JaphaVM No Tx	0.00	0.09	1.14

(a) Comparison of execution times for traversal 1.



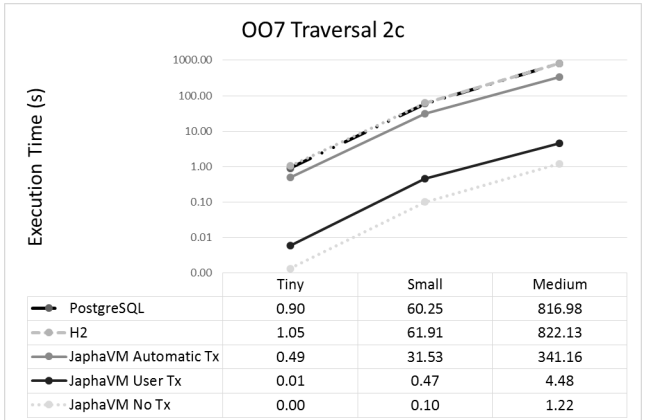
	Tiny	Small	Medium
PostgreSQL	0.92	59.69	821.95
H2	1.02	61.63	816.53
JaphaVM Automatic Tx	0.41	24.97	272.62
JaphaVM User Tx	0.01	0.39	3.64
JaphaVM No Tx	0.00	0.08	1.05

(b) Comparison of execution times for traversal 2a.



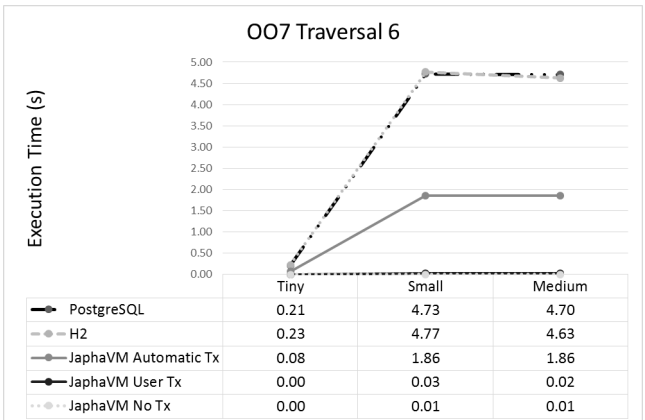
	Tiny	Small	Medium
PostgreSQL	0.93	60.25	853.28
H2	1.11	64.64	867.77
JaphaVM Automatic Tx	0.43	26.63	289.91
JaphaVM User Tx	0.01	0.43	3.99
JaphaVM No Tx	0.00	0.09	1.10

(c) Comparison of execution times for traversal 2b.



	Tiny	Small	Medium
PostgreSQL	0.90	60.25	816.98
H2	1.05	61.91	822.13
JaphaVM Automatic Tx	0.49	31.53	341.16
JaphaVM User Tx	0.01	0.47	4.48
JaphaVM No Tx	0.00	0.10	1.22

(d) Comparison of execution times for traversal 2c.



	Tiny	Small	Medium
PostgreSQL	0.21	4.73	4.70
H2	0.23	4.77	4.63
JaphaVM Automatic Tx	0.08	1.86	1.86
JaphaVM User Tx	0.00	0.03	0.02
JaphaVM No Tx	0.00	0.01	0.01

(e) Comparison of execution times for traversal 6.

Figure 5.2 – Curves with execution times for OO7 traversals across different db sizes (seconds, logarithmic scale)

Table 5.2 – I/O counters for *tiny* DB creation.

Counter	PostgreSQL DB	H2 DB	JaphaVM
read characters	3,208,732	3,517,082	36,060
written characters	405,182	577,291	499
read system calls	4,126	6,285	73
write system calls	1,602	3,763	16
# voluntary context switches	138	123	13
# involuntary context switches	1	1	1

Table 5.3 – I/O counters for *small* DB creation.

Counter	PostgreSQL DB	H2 DB	JaphaVM
read characters	9,102,770	17,936,590	82,105
written characters	11,150,783	19,702,246	74,490
read system calls	56,468	150,443	1,174
write system calls	53,958	147,928	1,116
# voluntary context switches	1231	1225	1104
# involuntary context switches	1	1	1

Table 5.4 – I/O counters for *medium* DB creation.

Counter	PostgreSQL DB	H2 DB	JaphaVM
read characters	75,888,918	172,910,579	82,540
written characters	129,663,311	243,547,947	74,490
read system calls	742,750	2,151,069	1,174
write system calls	741,264	2,148,581	1,116
# voluntary context switches	1209	1238	1104
# involuntary 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 Garbage Collection (GC). For the medium database size, all objects created by JaphaVM fit into the 8GB heap, not requiring GC; while the relational database scenarios required between 56—68 GCs (15—16 of them also performing heap compactions), which accounted for 15.62%—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, i.e. coarse-grained transactions (User Tx) have better performance than fine-grained transactions

(Automatic Tx), and not supporting transactions at all has the best performance.

Table 5.5 – I/O counters for Traversal 1 on *tiny* DB.

Counter	PostgreSQL DB	H2 DB	JaphaVM
read characters	3,590,297	4,393,436	29,636
written characters	539,701	1,047,596	504
read system calls	3,972	5,827	70
write system calls	1,421	3,320	16
# voluntary context switches	62	62	13
# involuntary context switches	1	1	1

Table 5.6 – I/O counters for Traversal 1 on *small* DB.

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

Table 5.7 – I/O counters for Traversal 1 on *medium* DB.

Counter	PostgreSQL DB	H2 DB	JaphaVM
read characters	384,046,805	803,662,022	86,127
written characters	308,148,130	553,984,993	75,953
read system calls	800,981	1,763,481	1,173
write system calls	798,418	1,760,978	1,180
# voluntary context switches	66	66	13
# involuntary context switches	1	1	1

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 5.5—5.7. 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 PostreSQL 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 (i.e., 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.

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, because its practical use is very limited 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.

On the next section, we describe another experiment, this time using a real-world application (Apache Lucene) instead of a synthetic benchmark.

5.2.3 Apache Lucene

We have created a modified version of the Apache Lucene search engine library [4] using orthogonal persistence 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 is a `java.util.HashSet` referencing the documents that contain that term (see Figure 5.3). 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 (e.g. e-mail search) and huge parallel machines (e.g. 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 relational databases using OO7.

Listing 5.3 – Code of Lucene OP inverted index

```
1 public class InvertedIndex {
2
3   protected Map<String, Set<Document>> invertedIndex;
4   protected int numberOfDocs = 0;
5   protected Map<Integer, Document> documents;
6
7   public InvertedIndex() {
8     invertedIndex = new HashMap<String, Set<Document>>();
9   }
10
11  public void addDocument(String term, Document doc) {
12    // add to term/doc map
13    Set<Document> documentSet = invertedIndex.get(term);
14    if (documentSet == null) {
15      documentSet = new HashSet<Document>();
16      invertedIndex.put(term, documentSet);
17    }
18    documentSet.add(doc);
19  }
20
21  public Set<Document> getDocsForTerm(String term) {
22    return invertedIndex.get(term);
23
24  }
25
26 }
```

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 [73] containing 680MB 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.

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 overhead is introduced by the journaled transactions mechanism used by NVML [27]. Before modifying the contents of a memory address for the first time (and only on the first time), it necessary to copy them to the undo log; in order to keep track of the address ranges that were already added to the undo log, NVML uses a radix tree. As more and more contiguous ranges are added to the undo log, the radix tree becomes deeper and traversal times increase. Since every data modification requires going through this process, writing transactionally to PM using an undo log is more expensive than reading.

In summary, this result indicates different trade-offs for the two scenarios: when accessing storage devices, writes typically perform better than reads; however, when accessing PM with journaled transactions (using an undo log), writes are more expensive than reads.

Table 5.8 – Lucene execution times (ms).

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 5.8 shows the execution time (in milliseconds) of each scenario using Lucene and Lucene-OP, the latter divided into user transactions or no transactions (for reference).

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 5.9), 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 5.8, we observe that they are more than one order of magnitude faster than the baseline version. We can also observe that since queries are read-most operations (as confirmed by the I/O counters on Tables 5.10 and 5.11), the transactional overhead has less impact than on the indexing task.

Table 5.9 – Lucene Indexing I/O counters.

Time (ms)	Baseline	Lucene-OP
read characters	543,801,327	426,434,910
written characters	229,084,470	62,716
read system calls	51,784	27,715
write system calls	15,334	1,209
# voluntary context switches	598	601
# involuntary context switches	987	355

Table 5.10 – Lucene single-term query I/O counters.

Time (ms)	Baseline	Lucene-OP
read characters	1,417,374	118,902
written characters	49,711	49,762
read system calls	1,972	640
write system calls	1,058	1,060
# voluntary context switches	527	527
# involuntary context switches	1,079	1,081

In the next section, we evaluate development complexity metrics for JaphaVM compared to traditional persistence approaches.

5.2.4 Development Complexity

It is a known fact that less complexity leads to reduced software development and maintenance effort, and improved quality [1, 47]. 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

Table 5.11 – Lucene double-term query I/O counters.

Time (ms)	Baseline	Lucene-OP
read characters	800,702	96,617
written characters	5,141	5,192
read system calls	869	166
write system calls	110	112
# voluntary context switches	53	54
# involuntary context switches	131	133

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. [11] 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 [10] 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.

Table 5.12 – Development complexity for OO7 scenarios.

Metric	OO7 JaphaVM	OO7 DB	Hibernate Framework
LOC	987	1,217	102,165
# Classes	18	19	1,340
# Methods	163	172	12,281

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 5.12 and Lucene's on Table 5.13.

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.

Table 5.13 – Development complexity for Lucene scenarios.

Metric	Lucene-OP	Lucene
LOC	1,067	8,314
# Classes	79	314
# Methods	247	1,513

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 5.13 shows that it requires seven times less LOCs, four times less classes and six times less methods to perform the same tasks.

In future works, we intend to do further research considering more experiments and metrics of development productivity, while expanding our sample of applications and programmers.

5.3 Evaluation Summary

This chapter has presented JaphaVM, a Java Virtual Machine designed for persistent memory and based on orthogonal persistence. JaphaVM's programming interface is inspired by previous research on orthogonally-persistent Java Virtual Machines, but its design leverages persistent memory, resulting in a simpler implementation with better performance. It uses an original combination of orthogonal persistence, persistent memory 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 experiments, JaphaVM executed the same operations between one and two orders of magnitude faster than the traditional implementations, with the exception of write-intensive tasks to files, and achieved the same results requiring significantly less lines of code, addressing research question 4, i.e., "*what is the performance of an orthogonally-persistent system using a persistent heap relative to traditional storage approaches?*".

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

5.4 Conclusion

In this chapter, we have presented JaphaVM, a prototype to validate and evaluate the effectiveness of the concepts proposed in Chapter 4. We have described the main design decisions behind JaphaVM, as well as experimental results concerning both execution performance and code complexity.

In the next chapter, we consider the present research in the context of related work.

6. RELATED WORK

In Section 3.4 we described several recent works proposing programming interfaces for persistent memory, including Mnemosyne [82], NV-Heaps [22], Atlas [20], Heapo [44], NVML [27], PyNVM [69], PCJ [28], MDS [34], and Apache Mnemonic [5]. All these works have in common the fact that persistent data are represented or allocated in a different way than volatile data. Mnemosyne, Atlas, HEAPO, NVML, PyNVM, and Apache Mnemonic require special memory allocators to be used for persistent data. NV-heaps, PCJ, MDS, and again Apache Mnemonic use specialized types for persistent objects. In addition, NV-heaps and NVML also use special pointers to refer to persistent objects.

Having a distinction between short- and long-lived data representations establishes an *impedance mismatch* that adds complexity to software development. The three principles of orthogonal persistence described in Section 2.1 try to address this problem by supporting a uniform treatment of objects irrespective of their types, allowing values of all types to have whatever longevity is required. By using special types, allocators or pointers for persistent data, all the programming interfaces described above do not conform to one or more of these principles.

Our design (described in Chapter 4), however, attempts to follow orthogonal persistence principles. Such principles can significantly increase programming and execution efficiency, as memory data structures are transparently persistent, without the need for programmatic persistence handling, and removing the need for crossing semantic boundaries.

JaphaVM, the vehicle for validating our design, can be compared with previous implementations of Java with orthogonal persistence, such as PJama [9], PEVM [51], ANU OPJ [57], Merpati [80], and Aspect-Oriented Programming implementations [2, 63, 76]. JaphaVM's programming interface approach is similar to these previous works, and has borrowed abstractions from them, especially persistence by reachability interfaces and callbacks for handling of external state. However, differently from JaphaVM, none of these systems was designed for persistent memory, so they still have to manage and move data across memory and storage, with all the entailed complexity and performance implications. PJama, PEVM, ANU OPJ have their own data stores implemented over binary files manipulated by special libraries. SoftPM and the AOP approach described in [2] support pluggable data stores, including databases and files. AOP implementations proposed in [63, 76] use relational databases. Marquez et al. [57] compared the performance of different versions of PJama and ANU-OPJ using the OO7 benchmark, and in all scenarios the introduction of orthogonal persistence resulted in overheads of one or two orders of magnitude when compared to a standard JVM. JaphaVM, on the other hand, is one or two orders of magnitude faster than traditional storage counterparts, due to its use of persistent memory, as discussed in Chapter 5.

Many programming interfaces for persistent memory use some form of persistence by reachability [20,22,27,34,40]. However, they define framework-specific persistence roots. JaphaVM leverages the persistence roots already used by Java's automatic memory management, satisfying the orthogonal

persistence principle of persistence identification, similarly to previous orthogonally-persistent Java interfaces [9, 51, 57]. We have chosen NVML as a lower-level layer for JaphaVM's failure-tolerant implementation, although others among the aforementioned C/C++ programming interfaces would be alternatives as well.

In summary, to the best of our knowledge, our work is the first to propose a design for languages with automatic memory management supporting orthogonal persistence principles and specific for persistent memory, based on an original combination of orthogonal persistence, persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions.

7. CONCLUSION

We started this thesis by posing our high-level research goal, namely to investigate the hypothesis that *persistent memory could be used to solve the performance and complexity issues historically associated with orthogonal persistence support in programming languages with automatic memory management*. We then refined this high level goal into different research questions, which were addressed by this thesis. We now summarize the answers and present our concluding remarks as well as possible directions for future work.

We first studied the evolution of Orthogonal Persistence concepts and systems, which attempted to alleviate the cost of mapping data as represented in memory to either files or databases (and vice-versa) by supporting a uniform treatment of objects irrespective of their types, allowing values of all types to have whatever longevity is required. We identified that previous implementations succeeded in providing simpler and easier programming interfaces, but had to cope internally with moving data between memory and storage. This challenge increased the integral system complexity, and led to performance constraints that ultimately prevented Orthogonal Persistence from becoming a mainstream approach.

Then, we studied recent non-volatile memory technologies that make it possible to collapse memory and storage into a single entity, known as persistent memory. We investigated several works adding Persistent Memory support for different programming languages, and identified that all of them provided simpler programming interfaces while still performing efficiently (since persistent memory removes the need for managing data movement between memory and storage). However, we also identified that none of the existing programming interfaces for persistent memory satisfied the orthogonal persistence goal of supporting a uniform treatment of objects independently of their types.

Based on these findings, we introduced a design for the runtime environment for languages with automatic memory management satisfying orthogonal persistence principles, based on an original combination of persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions. Such design can significantly increase programming and execution efficiency, as in-memory data structures are transparently persistent, without the need for programmatic persistence handling, and removing the need for crossing semantic boundaries.

In order to validate and evaluate the effectiveness of this design, we have created JaphaVM, a prototype Java Virtual Machine designed for persistent memory and based on Orthogonal Persistence. JaphaVM's programming interface is inspired by previous research on orthogonally-persistent Java Virtual Machines, but its design leverages Persistent Memory, resulting in a simpler implementation with better performance.

The initial prototype implementation of our design used a simple memory mapping to a file in persistent memory to store the heap and other runtime structures, without any PM-specific APIs to

handle in-memory transactions. Functional tests performed on this initial implementation answered affirmatively research question 1, i.e., that "*we can use a persistent heap on top of persistent memory to transparently store objects across multiple program executions*". No specific memory management features for persistent objects were implemented, using the standard garbage collection mechanisms, and the results of functional testing addressed research question 2, demonstrating that "*it is possible to leverage automatic memory management mechanisms to determine the persistence lifecycle of ordinary objects*".

However, consistency was not guaranteed in the presence of failures, as confirmed by tests where the prototype process was randomly killed. That led us to consider the use of lock-based failure-atomic transactions, i.e., using failure-atomic transactions (provided by libraries such as NVML) automatically at runtime. The final prototype implementation, using this mechanism, successfully addressed research question 3, demonstrating a way "*to automatically ensure the consistency of a persistent heap in the presence of failures*".

To address research question 4, i.e., "*what is the performance of an orthogonally-persistent system using a persistent heap relative to traditional storage approaches?*", we evaluated JaphaVM's advantages over relational databases and files, considering both execution performance and programming complexity. In our experiments, JaphaVM executed the same operations between one and two orders of magnitude faster than the traditional implementations, with the exception of write-intensive tasks to files, and achieved the same results requiring significantly less lines of code.

By addressing these different research questions, we believe to have successfully confirmed the hypothesis that *persistent memory can be used to solve the performance and complexity issues historically associated with orthogonal persistence support in programming languages with automatic memory management*.

7.1 Concluding Remarks

Based on the research work presented in this thesis, our main conclusion is that it is possible to leverage persistent memory to create orthogonally-persistent systems that are easy to both write and maintain, and have significantly superior performance. The present work has laid out conceptual foundations to design such systems for languages with automatic memory management, and has demonstrated it is technically feasible by means of a prototype implementation. Furthermore, we have shown through this work that there is great potential and value in investigating orthogonally-persistent programming interfaces for persistent memory.

Additionally, we list the main contributions (including technical and scientific contributions) of this work as follows:

- survey of persistent memory programming interfaces and evaluation against Orthogonal Persistence principles;
- proposal of a persistent heap adherent to orthogonal persistence concepts for languages with

automatic memory management, based on an original combination of persistent memory programming, persistence by reachability, and lock-based failure-atomic transactions;

- study of requirements and open challenges to achieve a robust orthogonally-persistent heap;
- creation of JaphaVM, a prototype Java Virtual Machine in order to validate and evaluate the proposed persistent heap design;
- study of JaphaVM's effectiveness and efficiency compared to traditional database- and file-based data stores using, respectively, OO7 and Apache Lucene workloads.

Lastly, from a more elevated perspective, we rate the present work as a tangible value case supporting the adoption of persistent memory and orthogonal persistence in order to create simpler programming interfaces with superior performance.

7.2 Future Research

There are many possible directions for future research based on this work. Firstly, our study identified many open challenges for the design of persistent heaps leveraging Persistent Memory. Secondly, JaphaVM, our prototype implementation, did not address some important concerns, such as type evolution and execution persistence. Lastly, the proposed design can be explored in other programming languages with automatic memory management beyond Java. We suggest some possibilities for further research as follows.

- Our persistent heap design can be extended to cover a) sharing data across different programs and programming languages, perhaps allowing a distributed, shared persistent heap; b) type evolution; c) mechanisms to detect and fix persistent bugs; d) abstractions for dealing with external state that are more integrated into the programming language and do not violate the persistence independence principle; e) security of data at rest and during computations.
- A deeper study of the effort required to write and maintain orthogonally-persistent applications in comparison to traditional database- and file-based persistence approaches. Atkinson et al. [11] stated in 1983 that about 30% of the lines of code were dedicated to persistence handling, and in a previous work [67] and in Section 5.2.4 of the present thesis we have made initial evaluations of the impact of persistent memory programming interfaces to development complexity. However, a broader and deeper study is necessary.
- The current version of JaphaVM can be evolved to address shortcomings, such as a) lack of execution persistence; b) lack of type evolution; c) not supporting Just-In-Time (JIT) compilation; and d) fixed heap size and address. Such evolution could allow broader investigations of the merits and shortcomings of its design.

- Explore the use of the proposed persistent heap design in other languages with managed runtimes and automatic memory management beyond Java, such as Python, Ruby, and JavaScript, among others.

Bibliography

[1] F. Akiyama. An example of software system debugging. In *Information Processing, vol. 71*, pages 353–379, 1971.

[2] M. Al-Mansari, S. Hanenberg, and R. Unland. Orthogonal persistence and aop: a balancing act. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*. ACM, 2007.

[3] S. Alagic and A. Yonezawa. Ambients of persistent concurrent objects. In *Proceedings of the 3rd International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, pages 155–161, 2011.

[4] Apache Software Foundation. Apache lucene. `https://lucene.apache.org/`, August 2016.

[5] Apache Software Foundation. Apache mnemonic. `https://github.com/apache/incubator-mnemonic`, March 2017.

[6] Apache Software Foundation. Apache mnemonic proposal. `https://wiki.apache.org/incubator/MnemonicProposal`, March 2017.

[7] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *DOOD*, volume 89, pages 40–57, 1989.

[8] M. Atkinson and M. Jordan. Issues raised by three years of developing pjama: An orthogonally persistent platform for java. In *Database Theory, ICDT99*, pages 1–30. Springer, 1999.

[9] M. Atkinson and M. Jordan. A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. *Sun Microsystems, Inc. Mountain View, CA, USA*, 2000.

[10] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–401, 1995.

[11] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The computer journal*, 26(4):360, 1983.

[12] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, 2011.

[13] R. Berryhill, W. Golab, and M. Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, number 20, pages 1–17, 2015.

[14] K. Bhandari, D. R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 677–694, New York, NY, USA, 2016. ACM.

[15] S. M. Blackburn and J. N. Zigman. Concurrency — the fly in the ointment? In *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems (POS-8) and the Third International Workshop on Persistence and Java (PJAVA-3), August 30-September 4, 1998, Tiburon, California*, page 259. Morgan Kaufmann, 1999.

[16] S. Bouchenak. Making java applications mobile or persistent. In *COOTS*, pages 159–172, 2001.

[17] R. Bruchhaus and R. Waser. Bipolar resistive switching in oxides for memory applications. In *Thin Film Metal-Oxides*, pages 131–167. Springer, 2010.

[18] G.W. Burr, B.N. Kurdi, J.C. Scott, C.H. Lam, K. Gopalakrishnan, and R.S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4):449–464, 2008.

[19] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data (SIGMOD '93), Peter Buneman and Sushil Jajodia (Eds.)*, pages 12–21. ACM, 1993.

[20] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[21] N. Christiansen. Windows Persistent Memory Support. In *Flash Memory Summit*. SNIA, 2016.

[22] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 105–118. ACM, 2011.

[23] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146. ACM, 2009.

[24] A. Cooper, R. Reimann, and H. Dubberly. *About face 2.0: the essentials of interaction design*. John Wiley & Sons, Inc., 2003.

[25] T. Cooper and M. Wise. Critique of orthogonal persistence. In *Proceedings of the 5th International Workshop on Object-Orientation in Operating Systems*, pages 122–126. IEEE, 1996.

[26] J. Corbet. Lfcs: Preparing linux for nonvolatile memory devices. `http://lwn.net/Articles/547903/`, April 2013.

[27] Intel Corp. Non-Volatile Memory Library (NVML). `http://pmem.io/nvml/`, August 2016.

[28] Intel Corp. Persistent Collections for Java. `https://github.com/pmem/pcj`, January 2017.

[29] A. Dearle and D. Hulse. Operating system support for persistent systems: past, present and future. *Software-Practice and Experience*, 30(4):295–324, 2000.

[30] A. Dearle, D. Hulse, and A. Farkas. Operating system support for java. In *Proceedings of the First International Workshop on Persistence and Java, Drymen, Scotland*, 1996. Captured in: http://blogs.cs.st-andrews.ac.uk/al/research/research-publications/, August 2011.

[31] A. Dearle, G. Kirby, and R. Morrison. Orthogonal persistence revisited. *Object Databases*, pages 1–22, 2010.

[32] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *Proceedings of the 45th ACM/IEEE Design Automation Conference (DAC 2008)*, pages 554–559. IEEE, 2008.

[33] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[34] Hewlett-Packard Enterprise. Managed Data Structures. `https://github.com/HewlettPackard/mds`, February 2017.

[35] A. Farkas and A. Dearle. Changing persistent applications. In M. Atkinson, D. Maier, and V. Benzaken, editors, *Persistent Object Systems*, Workshops in Computing, pages 302–315. Springer London, 1995.

[36] P. Ferreira and M. Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 394–401. IEEE, 1996.

[37] GNU Foundation. GNU Classpath. `http://http://www.gnu.org/s/classpath/`, January 2015.

[38] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[39] J. Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[40] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conference*, pages 319–331, 2012.

[41] J. Howell. Straightforward java persistence through checkpointing. *Advances in Persistent Object Systems*, pages 322–334, 1999.

[42] D. Hulse and A. Dearle. RT1R1/2: Report on the efficacy of persistent operating systems in supporting persistent application systems. *Research Paper, University of Stirling, Scotland*, 1999.

[43] D. Hulse, A. Dearle, and A. Howells. Lumberjack: A log-structured persistent object store. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*, pages 187–198. Morgan Kaufmann Publishers Inc., 1999.

[44] T. Hwang, J. Jung, and Y. Won. Heapo: Heap-based persistent object store. *ACM Transactions on Storage (TOS)*, 11(1):3, 2014.

[45] A. Ibrahim and W. R. Cook. Public implementation of oo7 benchmark. `https://sourceforge.net/projects/oo7/`, April 2013.

[46] M. Jordan and M. Atkinson. Orthogonal persistence for the java platform — specification. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2000.

[47] B. A. Kitchenham, L. M. Pickard, and S. J. Linkman. An evaluation of some design metrics. In *Software Engineering Journal 5(1)*, pages 50–58. Springer, 1990.

[48] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[49] M.H. Kryder and C.S. Kim. After Hard Drives - What Comes Next? *IEEE Transactions on Magnetics*, 45(10):3406–3413, 2009.

[50] B.C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[51] B. T. Lewis, B. Mathiske, and N. Gafter. Architecture of the pevm: A high-performance orthogonally persistent java [tm] virtual machine. Technical report, Sun Microsystems, Inc., 2000.

[52] D.L. Lewis and H.H.S. Lee. Architectural evaluation of 3D stacked RRAM caches. In *Proceedings of the IEEE International Conference on 3D System Integration (3DIC 2009)*, pages 1–4. IEEE, 2009.

[53] H. Li and Y. Chen. An overview of non-volatile memory technology and the implication for tools and architectures. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition 2009 (DATE'09)*, pages 731–736. IEEE, 2009.

[54] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013.

[55] R. Lougher. JamVM - a compact java virtual machine. `http://jamvm.sourceforge.net/`, June 2014.

[56] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Haallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, pages 50–58, 2002.

[57] A. Marquez, S. Blackburn, G. Mercer, and J. Zigman. Implementing orthogonally persistent java. In *Persistent Object Systems: Design, Implementation, and Use*, pages 247–261. Springer, 2001.

[58] A. Marquez, J.N. Zigman, and S.M. Blackburn. Fast portable orthogonally persistent java. *Software - Practice and Experience*, 30(4):449–479, 2000.

[59] J.C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+ DRAM hybrid main memory. In *Proceedings of the 12th conference on Hot topics in operating systems*, pages 14–14. USENIX Association, 2009.

[60] R. Morrison, R.C.H. Connor, G.N.C. Kirby, D.S. Munro, M.P. Atkinson, Q.I. Cutts, A.L. Brown, and A. Dearle. The Napier88 persistent programming language and environment. *Fully Integrated Data Environments*, pages 98–154, 1999.

[61] M. Oestreicher and K. Krishna. Object lifetimes in java card. In *USENIX Workshop on Smartcard Technology*. USENIX, 1999.

[62] A. O'Lenskie, A. Dearle, and D. Hulse. Orthogonally persistent support for persistent corba objects. Technical report, Technical Report 151, University of Stirling, Scotland, 2000.

[63] R.H.R. Pereira and J.B. Perez-Schofield. Orthogonal persistence and aop: a balancing act. In *Proceedings of the 6th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, 2011.

[64] T. Perez. Evaluation of system-level impacts of a persistent main memory architecture. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2012.

[65] T. Perez. JaphaVM code repository (GitHub). https://github.com/taciano-perez/JaphaVM, March 2017.

[66] T. Perez, N.L.V. Calazans, and C.A.F. De Rose. A preliminary study on system-level impact of persistent main memory. In *Proceedings of the 13rd International Symposium on Quality Electronic Design (ISQED 2012)*, pages 85–90. IEEE, 2012.

[67] T. Perez, N.L.V. Calazans, and C.A.F. De Rose. System-level impacts of persistent main memory using a search engine. *Microelectronics Journal*, 45(2):211–216, 2014.

[68] C. S. Perone. Documentation of python bindings for nvml library. http://pynvm.readthedocs.io/en/v0.1/, February 2016.

[69] C. S. Perone. Python bindings for nvml. https://github.com/perone/pynvm, February 2016.

[70] M. Piccioni, M. Oriol, and B. Meyer. Class schema evolution for persistent object-oriented software: Model, empirical study, and automated support. *IEEE Transactions on Software Engineering*, 39(2):184–196, 2013.

[71] Plexistor, https://sourceforge.net/projects/oo7/. *Plexistor SDM User Guide*, 12 2005.

[72] T. Printezis, M. Atkinson, L. Daynes, S. Spence, and P. Bailey. The design of a new persistent object store for pjama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, pages 61–74, 1997.

[73] Project Gutenberg. Project gutenberg aug 2003 cd. https://www.gutenberg.org/wiki/Gutenberg:The_CD_and_DVD_Project, August 2003.

[74] M.K. Qureshi, V. Srinivasan, and J.A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on computer architecture*, pages 24–33. ACM, 2009.

[75] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.H. Chen, H.L. Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2010.

[76] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM, 2003.

[77] Storage and Networking Industry Association (SNIA). *NVM Programming Model (NPM)*, 3 2015. Version 1.1.

[78] M. Stornelli. Protected and persistent ram filesystem. http://pramfs.sourceforge.net/, June 2013.

[79] D.B. Strukov, G.S. Snider, D.R. Stewart, and R.S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[80] T. Suezawa. Persistent execution state of a Java virtual machine. In *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*, pages 160–167, San Francisco, 2000. ACM Press.

[81] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 14. ACM, 2014.

[82] H. Volos, A.J. Tack, and M.M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 91–104. ACM, 2011.

[83] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *Proceedings of the 2006 European Conference on Object-Oriented Programming*, pages 148–173. Springer, 2006.

[84] M. Wilcox. Support ext4 on nv-dimms. http://lwn.net/Articles/588218/, February 2014.

[85] R.S. Williams. How we found the missing memristor. *IEEE spectrum*, 45(12):28–35, 2008.

[86] X. Wu and A.L. Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[87] J. Xu and S. Swanson. Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[88] J.J. Yang, M.D. Pickett, X. Li, D.A.A. Ohlberg, D.R. Stewart, and R.S. Williams. Memristive switching mechanism for metal/oxide/metal nanodevices. *Nature nanotechnology*, 3(7):429–433, 2008.

A. APPENDIX: JAPHAVM IMPLEMENTATION DETAILS

This appendix describes the JaphaVM prototype implementation in detail. The full source code can be found at [65].

A.1 JamVM Overview

JaphaVM is implemented using the open-source JamVM [55] Java Virtual Machine as baseline. In order to describe the modifications we have made to implement JaphaVM, first we need to provide an overview of how JamVMs works, which is done in this section.

JamVM runs on top of a wide set of Unix-like Operating Systems, including Linux, FreeBSD, OpenBSD, Solaris, OpenSolaris, Darwin, Android and iOS. It is ported to x86, x86-64, 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, which conforms to the JVM specification version 2 (blue book), was used as baseline for JaphaVM.

JamVM implements Ahead of Time Compilation (AOT). AOT is the act of compiling a high-level programming language (or intermediate, such as Java Bytecode) into native machine code. It is different from Just In Time (JIT) compilation, which is the compilation done during execution of a program (runtime) rather than prior to execution; JIT compilation is a combination of the two traditional approaches for translation to machine code: AOT and interpretation. The JamVM supports AOT but not JIT.

JamVM requires an implementation of the Java class libraries. We used the GNU Classpath [37] V0.92 open source implementation. Some modifications to GNU Classpath were made to address design issues, as described in the sections below.

A.1.1 JamVM Initialization

When a JamVM process is initiated, it performs a set of initialization steps, and then executes the `main()` method of a class specified as an argument during invocation. This sequence of steps is listed in Figure A.1

During execution, JamVM relies on a set of parameters that specify runtime behavior, including minimum and maximum size of heap, garbage collection parameters, among others. The first steps performed by JamVM populate these parameters with default values, and then replace the values of those passed as command line arguments.

The *Initialize Virtual Machine* step performs many essential activities necessary for the VM execution:

1. Initialize Platform.

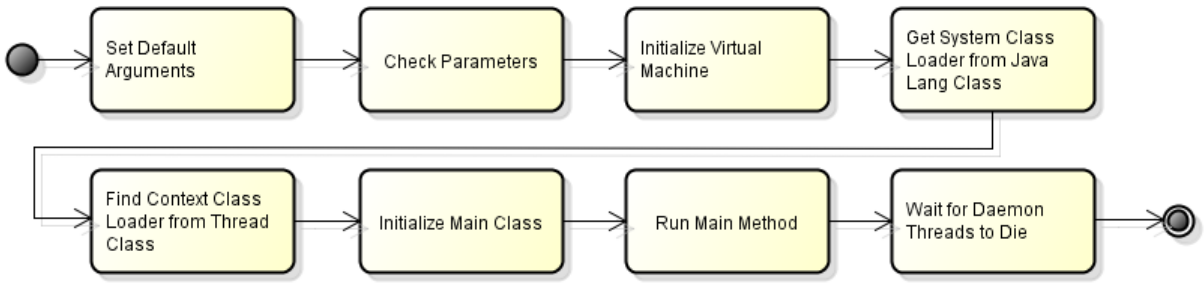


Figure A.1 – Overview of JamVM initialization steps

Detects the current platform and perform specific configurations.

2. Initialize Hooks.
   Register exit hooks, specifying functions that will be called upon VM termination.

3. Initialize Properties.
   Set user-defined command line properties.

4. Initialize Allocation.
   Allocate heap and set its base (heap data structures will be described in more detail below).

5. Initialize DLL.
   Allocate and initialize the DLL hash table.

6. Initialize Symbols Hash Table.
   Allocate and initialize a hash table used to index several data structures. The keys are strings encoded using UTF-8 and the values are pointers to the appropriate structures.

7. Initialize Threads - stage 1.
   Set stack size and establish locks utilized internally by the JamVM.

8. Initialize Symbols.
   Add symbols to the symbols hash table.

9. Initialize Classes.
   Load and initialize boot loader class.

10. Initialize Monitors.
    Allocate and initialize Java monitors hash table.

11. Initialize String.
    Load and initialize String class.

12. Initialize Exception.
    Load and register Exception classes.

13. Initialize Natives.
    Initialize protection domains.

14. Initialize JNI.
    Initialize the global reference tables, cache classes and methods/fields for JNI 1.4 NIO support.

15. Initialize interpreter.
    Start interpreter.

16. Initialize Threads - stage 2
    Start main thread.

17. Initialize garbage collector.
    Initialize OOM (Out Of Memory) class and start finalizer and handler threads.

After initialization activities are finalized, the JamVM locates the appropriate class loader, loads the main class and invokes its `main()` method.

A.1.2 Internal Data Structures

The data structures that are more important to our purposes are the ones storing object instances (heap), type (class/interface) definitions, threads, and thread stacks.

Heap

The JamVM heap stores object instances (including arrays) and type definitions. the `mmap()` call is used in order to allocate a contiguous memory space. The heap has an initial minimum size and may grow up to a maximum value. Both minimum and maximum values are configurable.

In order to manage allocated/free space inside the heap, data is organized by chunks. Chunks are indexed by a linked list called `freelist` and are ordered by their place on the heap. This list is often manipulated by the Garbage Collector (GC) in order to reclaim free space.

During JamVM initialization, a chunk covering the entire heap is created. New chunks are created during runtime when loading classes and instantiating objects. Allocation of chunks inside the heap is made by the `gcMalloc()` and `gcRealloc()` functions. Release of heap space is made by the GC. The `freelist` is updated every time GC or the allocation functions are executed. The chunk list inside the heap is depicted in Figure A.2.

Chunks have a simple structure consisting of a header, a pointer to the next chunk, and the chunk's contents. The header layout is depicted in Figure A.3

Description of header fields:

- 31: Has hash code bit
- 30: Hash code taken bit

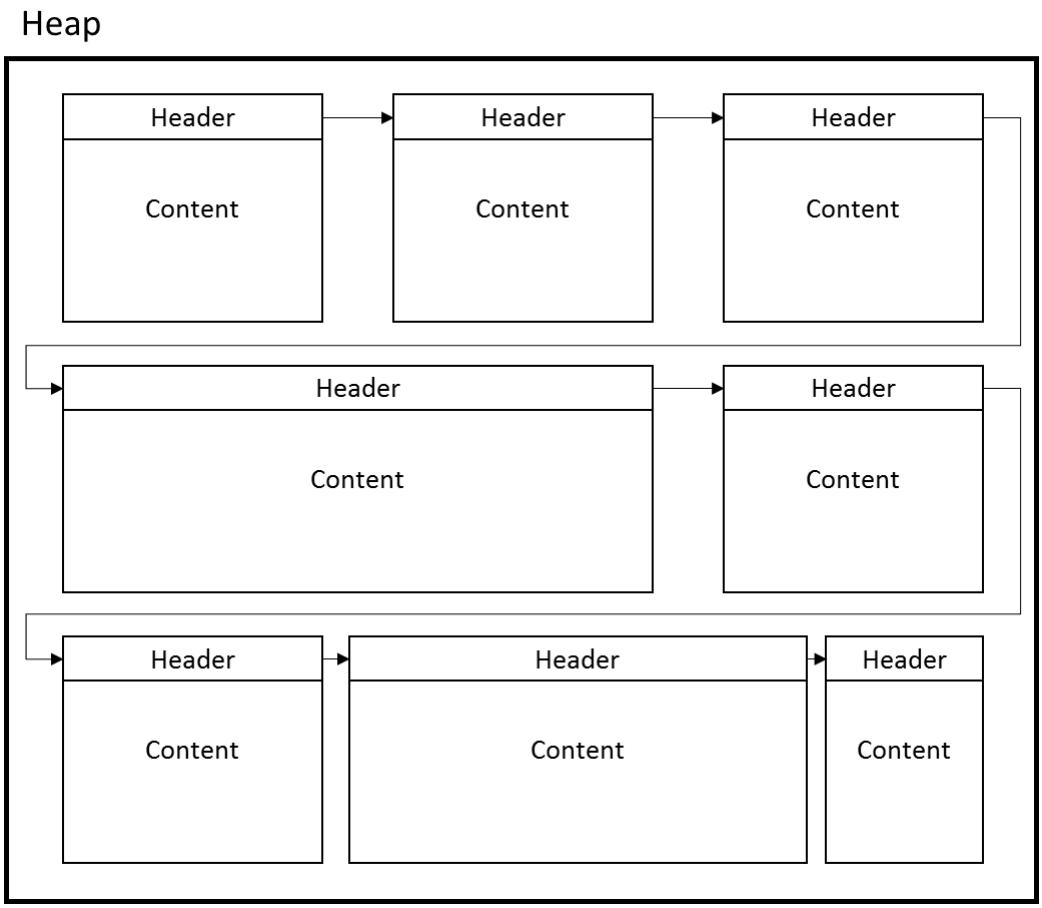


Figure A.2 – Chunks organization inside heap

Figure A.3 – Chunk header format

- 29-3(Block Size): Quantity of allocated bytes
- 2: Is special bit
- 1: FLC bit
- 0: Allocated bit

Allocation/release of data inside the heap is made through the functions `gcMalloc()`, `gcRealloc()`, and `gcFree()`.

Memory allocation outside the heap is generally made using `sysMalloc()`, `sysFree()`, `sysRealloc()`. Its implementation uses standard `malloc()`, `free()` and `realloc()` C functions.

Types and Objects

Each definition of a Java type (class or interface) consists of a `Classblock` associated with a set of `Fieldblocks` and a set of `Methodblocks`. `Classblocks` are allocated inside the heap, while `Fieldblocks` and `Methodblocks` are allocated outside of the heap area. Figure A.4 depicts these data structures and their relationships.

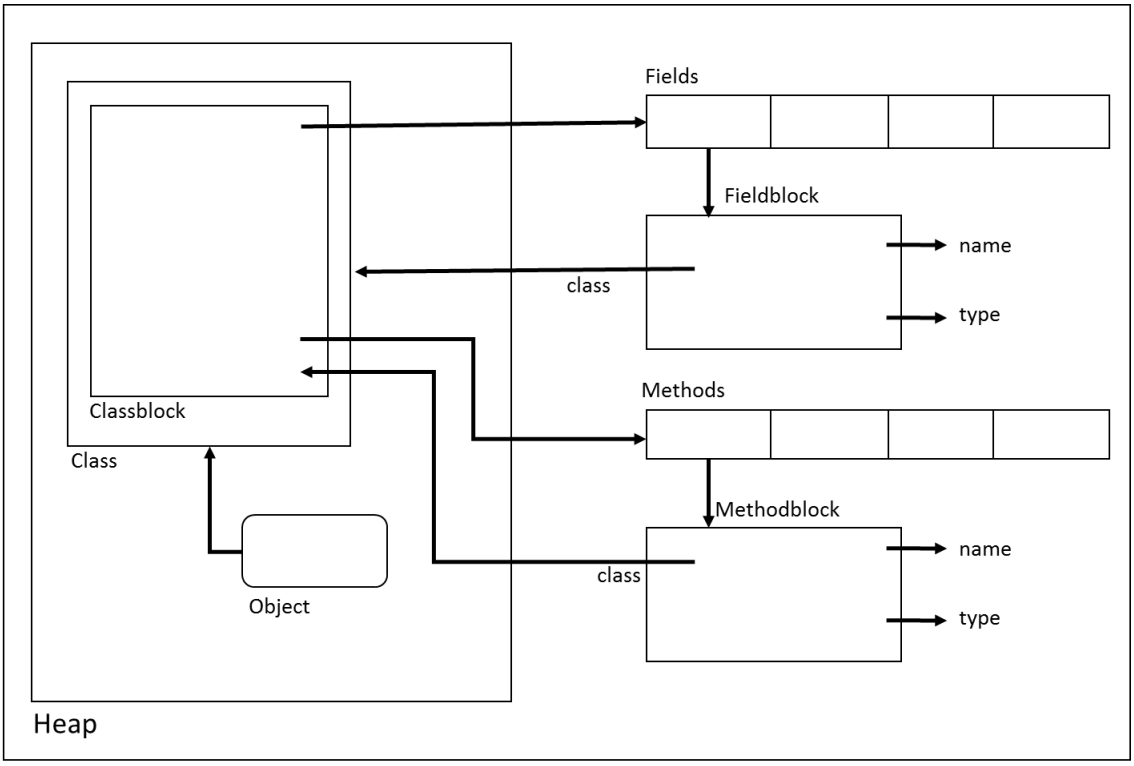


Figure A.4 – Class organization in JamVM memory

Types and objects are implemented using the same `object` struct presented in code listing A.1. The `*class` pointer references the `Classblock` address inside the heap + 1 (32-bit chunk header offset). Every object references its type.

Listing A.1 – Class structure

```
1 typedef struct object Class;
2
3 typedef struct object {
4    uintptr_t lock;
5    Class *class;
6 } Object;
```

Classblock

The Classblock is the root structure of a type, holding references to all other structures. Its definition can be viewed in code listing A.2.

Listing A.2 – Classblock structure

```
1  typedef struct classblock {
2     uintptr_t pad[CLASS_PAD_SIZE];
3     char *name;
4     char *signature;
5     char *super_name;
6     char *source_file_name;
7     Class *super;
8     u1 state;
9     u2 flags;
10    u2 access_flags;
11    u2 interfaces_count;
12    u2 fields_count;
13    u2 methods_count;
14    u2 constant_pool_count;
15    int object_size;
16    FieldBlock *fields;
17    MethodBlock *methods;
18    Class **interfaces;
19    ConstantPool constant_pool;
20    int method_table_size;
21    MethodBlock **method_table;
22    int imethod_table_size;
23    ITableEntry *imethod_table;
24    Class *element_class;
25    int initing_tid;
26    int dim;
27    Object *class_loader;
28    u2 declaring_class;
29    u2 inner_access_flags;
30    u2 inner_class_count;
31    u2 *inner_classes;
32    int refs_offsets_size;
33    RefsOffsetsEntry *refs_offsets_table;
34    u2 enclosing_class;
35    u2 enclosing_method;
36    AnnotationData *annotations;
37 } ClassBlock;
```

Fieldblock

In lines 12 and 16 of code listing A.2, variable `fields_count` and the `fields` array are declared. When a class is loaded and defined in JamVM the `fields` array points to `FieldBlock` structs with `fields_count` positions. These structs are allocated outside of the heap, and hold all class attributes.

The `Fieldblock` definition can be viewed in code listing A.3.

Listing A.3 – Fieldblock struct

```
typedef struct fieldblock {
    Class *class;
    char *name;
    char *type;
    char *signature;
    u2 access_flags;
    u2 constant;
    AnnotationData *annotations;
    union {
        union {
            char data[8];
            uintptr_t u;
            long long l;
            void *p;
            int i;
        } static_value;
        u4 offset;
    } u;
} FieldBlock;
```

Methodblock

In lines 13 and 17 of code listing A.2, variable `methods_count` and array `methods` array, are declared. When a class is loaded and defined in JamVM, array `methods` references `Methodblock` structs with `methods_count` positions. These structs are allocated outside of the heap, and hold all class attributes.

The `Methodblock` implementation can be viewed in code listing A.4.

Listing A.4 – Methodblock struct

```
struct methodblock {
    Class *class;
    char *name;
    char *type;
    char *signature;
    u2 access_flags;
    u2 max_stack;
    u2 max_locals;
    u2 args_count;
    u2 throw_table_size;
    u2 exception_table_size;
    u2 line_no_table_size;
    int native_extra_arg;
    NativeMethod native_invoker;
    void *code;
    int code_size;
    u2 *throw_table;
```

```
18    ExceptionTableEntry *exception_table;
19    LineNoTableEntry *line_no_table;
20    int method_table_index;
21    MethodAnnotationData *annotations;
22 #ifdef INLINING
23    QuickPrepareInfo *quick_prepare_info;
24    ProfileInfo *profile_info;
25 #endif
26 };
```

Constant Pool

In line 14 and 19 of code listing A.2, variable `constant_pool_count` and array `constant_pool` are defined. When a class is loaded and defined in JamVM, a table of `constant_pool_count * sizeof(ConstantPoolEntry)` is allocated outside of the heap to hold the constant pool table of a type. The definition of `ConstantPoolEntry` showed in code listing A.5 follows its standard definition suggested by the Java Virtual Machine Specification [54].

Listing A.5 – ConstantPoolEntry structure

```
1 typedef uintptr_t ConstantPoolEntry;
2
3 typedef struct constant_pool {
4     volatile u1 *type;
5     ConstantPoolEntry *info;
6 } ConstantPool;
```

A.1.3 Threads and Thread Stacks

The loading and initialization of Thread classes are performed in distinct steps by the JamVM. This happens because the Class Loader needs to be executed in a valid thread and execution environment, but in order to create and initialize a Thread it is required to load the Thread class.

In the loading step, classes are loaded from their `.class` files, but the initializers, static blocks and constructors are not executed.

In the initialization step, after the Thread class is loaded, the initializers, and static blocks and constructors are executed, allowing the main thread to proceed.

JamVM threads are allocated outside the heap. All threads allocate at least 1 MB for stack space. The `thread` struct presented on code listing A.6 holds all thread-related information.

Listing A.6 – Thread structure

```
1 struct thread {
2     int id;
3     pthread_t tid;
4     char state;
5     char suspend;
```

```
6       char blocking;
7       char park_state;
8       char interrupted;
9       char interrupting;
10      ExecEnv *ee;
11      void *stack_top;
12      void *stack_base;
13      Monitor *wait_mon;
14      Monitor *blocked_mon;
15      Thread *wait_prev;
16      Thread *wait_next;
17      pthread_cond_t wait_cv;
18      pthread_cond_t park_cv;
19      pthread_mutex_t park_lock;
20      long long blocked_count;
21      long long waited_count;
22      Thread *prev, *next;
23      unsigned int wait_id;
24      unsigned int notify_id;
25  };
```

In line 10, an Execution Environment is declared. The `ExecEnv` struct can be viewed on code listing A.7.

Listing A.7 – ExecutionEnvironment structure

```
1  typedef struct exec_env {
2      Object *exception;
3      char *stack;
4      char *stack_end;
5      int stack_size;
6      Frame *last_frame;
7      Object *thread;
8      char overflow;
9  } ExecEnv;
```

A.1.4 Symbols Hash Table

JamVM uses indices implemented as hash tables to keep track and perform quick searches of different entries on the JamVM environment.

The hash tables used by JamVM are allocated outside the heap, using the `gcMemMalloc()`, `gcMemFree()`, and `gcMemRealloc()` functions.

All hash tables have a generic structure as depicted on code listing A.8 and a generic function for finding, adding and resizing as depicted on code listing A.9.

Listing A.8 – HashTable structs

```
typedef struct hash_entry {
    void *data;
    int hash;
} HashEntry;

typedef struct hash_table {
    HashEntry *hash_table;
    int hash_size;
    int hash_count;
    VMLock lock;
} HashTable;
```

Listing A.9 – findHashEntry function

```
#define findHashEntry(table, ptr, ptr2, add_if_absent, scavenge, locked)
{
    int hash = HASH(ptr);
    int i;

    Thread *self;
    if(locked)
    {
        self = threadSelf();
        lockHashTable0(&table, self);
    }

    i = hash & (table.hash_size - 1);

    for(;;)
    {
        ptr2 = table.hash_table[i].data;
        if((ptr2 == NULL) || (COMPARE(ptr, ptr2, hash, table.hash_table[i].
            hash)))
            break;

        i = (i+1) & (table.hash_size - 1);
    }

    if(ptr2)
    {
        ptr2 = FOUND(ptr, ptr2);
    } else
        if(add_if_absent)
        {
            table.hash_table[i].hash = hash;
            ptr2 = table.hash_table[i].data = PREPARE(ptr);
```

```
32
33              if(ptr2)
34              {
35                table.hash_count++;
36                  if((table.hash_count * 4) > (table.hash_size * 3))
37                  {
38                      int new_size;
39                      if(scavenge)
40                      {
41                          HashEntry *entry = table.hash_table;
42                          int cnt = table.hash_count;
43                          for(; cnt; entry++)
44                          {
45                              void *data = entry->data;
46                              if(data)
47                              {
48                                  if(SCAVENGE(data))
49                                  {
50                                      entry->data = NULL;
51                                      table.hash_count--;
52                                  }
53                                  cnt--;
54                              }
55                          }
56                          if((table.hash_count * 3) > (table.hash_size * 2))
57                          {
58                              new_size = table.hash_size*2;
59                          }
60                          else
61                          {
62                              new_size = table.hash_size;
63                          }
64                      }
65                      else
66                      {
67                          new_size = table.hash_size*2;
68                      }
69                      resizeHash(&table, new_size);
70                  }
71              }
72          }
73      if(locked)
74      {
75          unlockHashTable0(&table, self);
76      }
77  }
```

Each hash table has a specific function for hash creation and comparison.
JamVM has the following hash tables:

- Symbols hash table — contains symbols encoded with UTF-8. Hash creation and comparisons are made using UTF-8 strings.
- Classes hash table — contains classes pointers to the heap. Hash creation and comparisons are made using UTF-8 strings.
- Boot Packages hash table — contains packages loaded by the boot loader. Hash creation and comparisons are made using UTF-8 strings.
- Boot classes hash table — contains classes loaded by the boot loader and internally created arrays. Hash creation and comparisons are made using UTF-8 strings.
- Strings hash table — contains Java strings. Hash creation and comparisons are made using string data.
- DLL hash table — Contains file descriptors to external structures (e.g. Operational system files). Hash creation and comparisons are made using file names.

A.2 Changes Required to Implement JaphaVM

As expected, most changes made to implement JaphaVM are related to memory management, such as persistence of heap structures, data referenced by heap structures, and JamVM hash tables. Additional changes were made to the GNU Classpath Java class library to address the handling of external state (such as standard console input/output).

The following sections present all changes made to the JamVM baseline in order to implement the JaphaVM prototype.

A.2.1 Invocation Arguments

In order 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 non-persistent mode as depicted on code listing A.10. The persistent heap file name must be informed as a parameter. On the initialization functions, a global boolean variable called `persistent` is created in order to verify the execution mode throughout the code.

Listing A.10 – New invocation argument (jam.c)

```
1 ...
2       args->persistent_heap = FALSE;
3 ...
4               if (strncmp(argv[i], "-persistentheap:", 16) == 0) {
5                     args->persistent_heap = TRUE;
```

```
6                       args->heap_file = argv[i] + 16;
7                       persistent = TRUE;
8               }
```

Invoking the virtual machine with the `-persistentheap` flag will execute in persistent heap mode. Otherwise, it will execute the standard JamVM, simplifying tests and comparisons.

A.2.2 Heap Persistence

As previously mentioned, the original JamVM implementation uses a private memory mapped area via the `mmap` function in order to allocate memory for the JVM heap.

Our initial implementation of the persistent heap only modified the `mmap` flags to use a shared memory mapping using a file on top of a PM-aware file system. Although this implementation indeed worked to provide heap persistence, it was not fault-tolerant, and unexpected application termination could leave the persistent heap in an inconsistent state.

The final JaphaVM implementation uses the NVML library [27] in order to provide fault-tolerance, as NVML provides journaled transactions via an undo log in PM.

We introduced a data structure that represents the persistent heap and contains pointers to its data and metadata. The `PHeap` data structure is shown on Listing A.11.

Listing A.11 – PHeap structure (jam.h)

```
1  typedef struct pheap {
2    void *base_address;
3    Chunk *freelist;
4    Chunk **chunkpp;
5    unsigned long heapfree;
6    unsigned long maxHeap;
7    char *heapbase;
8    char *heapmax;
9    char *heaplimit;
10   nvmChunk *nvmfreelist;
11   nvmChunk **nvmChunkpp;
12   unsigned int nvmFreeSpace;
13   unsigned int nvmCurrentSize;
14   unsigned long nvm_limit;
15   OPC opc;
16   char* utf8_ht[UTF8_HT_SIZE];
17   char* bootCl_ht[BOOTCL_HT_SIZE];
18   char* bootPck_ht[BOOTPCK_HT_SIZE];
19   char* string_ht[STRING_HT_SIZE];
20   char* classes_ht[CLASSES_HT_SIZE];
21   char* monitor_ht[MONITOR_HT_SIZE];
22   char* zip_ht[ZIP_HT_SIZE];
23   char nvm[NVM_INIT_SIZE];
24   char heapMem[HEAP_SIZE];// heap contents
```

```
25 } PHeap;
```

`PHeap` member `heapMem` provisions the actual heap allocation space. Its size is currently defined by `HEAP_SIZE` at compile time and is not dynamically resizable.

The member `base_address` exists to guarantee that the persistent heap will be mapped to the same base address across multiple executions of the JVM. This is necessary to ensure that persistent pointers will continue pointing to the right addresses. It would be possible to avoid the need for using the same base address by using relocatable pointers provided by NVML (`PMEMoid`s). However, that would require the modification of all JamVM opcodes. For this experiment, we decided for simplicity reasons to use absolute pointers and require the persistent heap to be always remapped to the same base address. As consequence, before executing the JVM it is necessary to set the NVML environment variable `PMEM_MMAP_HINT` to the desired base address.

`Pheap` members `heapfree`, `maxHeap`, `heapbase`, `heapmax`, and `heaplimit` hold heap metadata. Members `freelist` and `chunkpp` contain pointers to heap memory chunks, as described in Section A.1.2.

Pheaap members `nvmfreelist`, `nvmChunkpp`, `nvmFreeSpace`, `nvmCurrentSize`, and `nvm_limit` contain pointers for a separate chunk area specific for non-heap metadata, provisioned in member `nvm`. The size of the latter is defined by `NVM_INIT_SIZE` at compile time and is not dynamically resizable. Non-heap metadata are described in further detail in Section A.2.7.

Members ending with `_ht` hold pointers to internal hash tables, described in more detail in Section A.2.8.

The member `opc` contains a pointer to an `OPC` data structure containing additional metadata used for managing internal VM hash tables and Garbage Collection, among others. The OPC struct is listed on Listing A.12.

Listing A.12 – OPC struct

```
1 typedef struct opc {
2   int ldr_vmdata_offset;
3   Class *java_lang_Class;
4   unsigned long chunkpp;
5   uintptr_t freelist_header;
6   struct chunk *freelist_next;
7   unsigned int heapfree;
8   int boot_classes_hash_count;
9   int boot_packages_hash_count;
10  int classes_hash_count;
11  int string_hash_count;
12  int utf8_hash_count;
13  Class *prim_classes[9];
14  Object **has_finaliser_list;
15  int has_finaliser_count;
16  int has_finaliser_size;
17 } OPC;
```

A.2.3 NVML Transactions

As mentioned previously, we have chosen to use the NVML library in order to make persistent heap operations fault-tolerant by means of transactions.

In order to have proper transactional behavior using NVML, we need to start/end transactions at the appropriate points, and we need to ensure that all data about to be modified is added to the undo log before the actual modification.

We have created three macros to define the start and the end of a execution as well as a macro to add data to the undo log, depicted on A.13. These macros are used every time data inside the NVML Memory Pool is modified.

Listing A.13 – NVML transaction macros

```
1   uint total_tx_count;
2   #define NVML_DIRECT(TYPE, PTR, SIZE) if(pmemobj_tx_stage() ==
        TX_STAGE_WORK) { \
3                         if(errr = pmemobj_tx_add_range_direct(PTR, SIZE)) { \
4                           printf("%s ERROR %d: could not add range to
                                transaction\n", TYPE, errr); \
5                         } \
6                       }
7
8   #define BEGIN_TX(TYPE) if(errr = pmemobj_tx_begin(pop_heap, NULL,
        TX_LOCK_NONE)) { \
9                        printf("ERROR %d at BEGIN\n", errr); \
10                             } else { \
11                    total_tx_count++; \
12                  } \
13
14  #define END_TX(TYPE) if(pmemobj_tx_stage() == TX_STAGE_WORK) { \
15                    pmemobj_tx_process(); \
16                } \
17                if(pmemobj_tx_stage() != TX_STAGE_NONE) { \
18                  flushPHValues(); \
19                  pmemobj_tx_end(); \
20                    total_tx_count--; \
21                }
```

A.2.4 NVML Memory Pool Initialization

In order to access data in PM via NVML, it is necessary to declare a Memory Pool. The memory pool declaration is depicted on Listing A.14.

Listing A.14 – NVML Memory Pool declaration (jam.h)

```
1 POBJ_LAYOUT_BEGIN(HEAP_POOL);
2 POBJ_LAYOUT_ROOT(HEAP_POOL, PHeap);
3 POBJ_LAYOUT_END(HEAP_POOL);
4
5 int nvml_alloc, persistent, errr, first_ex;
6 PMEMobjpool *pop_heap;
7 PMEMoid root_heap;
8 PHeap *pheap;
```

Once the memory pool is defined, it needs to be created by means of the NVML function `pmemobj_create()`. In posterior executions of the VM, it can be reopened with `pmemobj_open()`. Both operations are shown on Listing A.15.

Listing A.15 – NVML File initialization method (alloc.c)

```
1 int initialiseRoot(InitArgs *args) {
2   unsigned long heap_size;
3
4   if(HEAP_SIZE < PMEMOBJ_MIN_POOL)
5     heap_size = PMEMOBJ_MIN_POOL;
6   else
7     heap_size = HEAP_SIZE;
8
9   if(access(PATH, F_OK) != 0) {
10    if((pop_heap = pmemobj_create(PATH, POBJ_LAYOUT_NAME(HEAP_POOL),
          PHEAP_SIZE, 0666)) == NULL) {//8388608
11      printf("failed to create pool\n");
12      printf("error msg:\t%s\n", pmem_errormsg());
13      return FALSE;
14    }
15    BEGIN_TX("INITIALISEROOT CREATING")
16    root_heap = pmemobj_root(pop_heap, heap_size);
17    pheap = (PHeap*) pmemobj_direct(root_heap);
18    pheap->base_address = pheap;
19    pheap->heapfree = HEAP_SIZE - sizeof(Chunk);
20    pheap->maxHeap = HEAP_SIZE;
21    pheap->heapbase = (char*) (((uintptr_t)pheap->heapMem + HEADER_SIZE +
          OBJECT_GRAIN-1) & ~(OBJECT_GRAIN-1)) - HEADER_SIZE;
22    pheap->heapmax = pheap->heapbase + ((args->max_heap - (pheap->heapbase -
          pheap->heapMem)) & ~(OBJECT_GRAIN - 1));
23    pheap->heaplimit = pheap->heapbase + ((args->min_heap - (pheap->heapbase
          - pheap->heapMem)) & ~(OBJECT_GRAIN - 1));
24    pheap->freelist = (Chunk*) pheap->heapbase;
25    pheap->freelist->header = pheap->heaplimit - pheap->heapbase;
26    pheap->freelist->next = NULL;
27    pheap->chunkpp = &(pheap->freelist);
28  }
```

```
29    else {
30      if((pop_heap = pmemobj_open(PATH, POBJ_LAYOUT_NAME(HEAP_POOL))) == NULL)
            {
31        printf("failed to open pool\n");
32        return FALSE;
33      }
34      BEGIN_TX("INITIALISEROOT OPENING")
35      root_heap = pmemobj_root(pop_heap, heap_size);
36      pheap = (struct pheap*) pmemobj_direct(root_heap);
37      if(pheap->base_address != pheap) {
38        printf("ERROR: base addresses are different, will abort VM execution\n"
              );
39        printf("Expected base address was %p, but current base address is %p\n"
              , pheap->base_address, pheap);
40        //pmemobj_close(pop_heap);  // attempt to close memory pool is
              generating segfault, so we'll just skip it
41        exit(-1);
42      }
43    }
44
45    return TRUE;
46  }
```

A.2.5 VM Initialization

The whole VM initialization process happens inside a single NVML transaction. This way, if the VM is interrupted unexpectedly in the middle of initialization, it will not leave a corrupt Memory Pool behind. This transactions starts when the Memory Pool is opened (Listing A.15) and ends when the `initVM` function finishes (see Listing A.16).

Listing A.16 – InitVM function (init.c)

```
1  void initVM(InitArgs *args) {
2      /* Perform platform dependent initialisation */
3      initialisePlatform();
4
5      nvml_alloc = TRUE;
6
7      initialiseHooks(args);
8      initialiseProperties(args);
9      initialiseAlloc(args);
10     initialiseUtf8(args);
11     initialiseThreadStage1(args);
12     initialiseSymbol();
13     initialiseClass(args);
14     initialiseMonitor(args);
```

```
15      initialiseString(args);
16      initialiseException();
17      initialiseDll(args);
18      initialiseNatives();
19      initialiseJNI();
20      initialiseInterpreter(args);
21      initialiseThreadStage2(args);
22      initialiseGC(args);
23
24      END_TX("INITVM")
25      nvml_alloc = VM_initing = FALSE;
26  }
```

A.2.6 Persistent Heap Allocation

The function responsible for heap allocation in the original JamVM code is `gcMalloc`. Our implementation introduces the `ph_malloc` function that is used when the persistent heap is enabled. This function manipulates heap chunks in the Memory Pool in transactional context. The `ph_malloc` function is listed on Listing A.17.

Listing A.17 – ph_malloc function (alloc.c)

```
1  void *ph_malloc(int len2) {
2    uintptr_t largest;
3    Chunk *found;
4    Thread *self;
5    int err;
6    int have_remaining = FALSE;
7
8    char *ret_addr;
9
10   int n = (len2+HEADER_SIZE+OBJECT_GRAIN-1)&~(OBJECT_GRAIN-1);
11   /* Grab the heap lock, hopefully without having to
12      wait for it to avoid disabling suspension */
13   self = threadSelf();
14   if(!tryLockVMLock(heap_lock, self)) {
15     disableSuspend(self);
16     lockVMLock(heap_lock, self);
17     enableSuspend(self);
18   }
19
20   /* Scan freelist looking for a chunk big enough to
21      satisfy allocation request */
22   int has_found = FALSE;
23
24   if(nvml_alloc) {
```

```
25      NVML_DIRECT("CHUNKPP", &pheap->chunkpp, sizeof(Chunk*))
26    }
27
28    for (;;) {
29
30      uintptr_t len;
31      while(*(pheap->chunkpp)) {
32        len = (*(pheap->chunkpp))->header;
33        if(len == n) {
34          found = *pheap->chunkpp;
35          *pheap->chunkpp = found->next;
36          has_found = TRUE;
37          goto got_it_pmem;
38        }
39        if(len > n) {
40          Chunk *rem;
41          found = *pheap->chunkpp;
42          rem = (Chunk*)((char*)found + n);
43          rem->header = len - n;
44
45          /* Chain the remainder onto the freelist only
46             if it's large enough to hold an object */
47          if(rem->header >= MIN_OBJECT_SIZE) {
48            have_remaining = TRUE;
49            rem->next = found->next;
50            *pheap->chunkpp = rem;
51          } else
52            *pheap->chunkpp = found->next;
53
54          has_found = TRUE;
55          goto got_it_pmem;
56        }
57        pheap->chunkpp = &(*pheap->chunkpp)->next;
58      }
59
60      if (!has_found) {
61        if(verbosegc) jam_printf("<GC: Alloc attempt for %d bytes failed.>\n",
            n);
62
63        /* The state determines what action to take in the event of
64        allocation failure.  The states go up in seriousness,
65        and are visible to other threads */
66        static enum { gc, run_finalizers, throw_oom } state = gc;
67
68        switch(state) {
69
70          case gc:
```

```
71            /* Normal failure.  Do a garbage-collection and retry
72               allocation if the largest block satisfies the request.
73               Attempt to ensure heap is at least 25% free, to stop
74               rapid gc cycles */
75            largest = gc0_pmem(TRUE, FALSE);
76
77            if(n <= largest && (pheap->heapfree * 4 >= (pheap->heaplimit -
                 pheap->heapbase)))
78              break;
79
80            /* We fall through into the next state, but we need to set
81               the state as it will be visible to other threads */
82            state = run_finalizers;
83
84          case run_finalizers:
85            /* Before expanding heap try to run outstanding finalizers.
86               If gc found new finalizers, this gives the finalizer chance
87               to run them */
88            /* XXX NVM CHANGE 000.000.000  */
89            //msync(heapMem, maxHeap, MS_SYNC);
90            unlockVMLock(heap_lock, self);
91            disableSuspend(self);
92
93            if(verbosegc)
94              jam_printf("<GC: Waiting for finalizers to be ran.>\n");
95
96            runFinalizers0(self, 200);
97            lockVMLock(heap_lock, self);
98            enableSuspend(self);
99
100           if(state != run_finalizers)
101             break;
102
103           /* Retry gc, but this time compact the heap rather than just
104              sweeping it */
105           largest = gc0_pmem(TRUE, TRUE);
106           if(n <= largest && (pheap->heapfree * 4 >= (pheap->heaplimit -
                 pheap->heapbase))) {
107             state = gc;
108             break;
109           }
110
111           /* Still not freed enough memory so try to expand the heap.
112              Note we retry allocation even if the heap couldn't be
113              expanded sufficiently -- there's a chance gc may merge
114              adjacent blocks together at the top of the heap */
115              /*
```

```
116             if(heaplimit < heapmax) {
117               expandHeap(n);
118               state = gc;
119               break;
120             }
121             */
122
123             if(verbosegc)
124               jam_printf("<GC: Stack at maximum already."
125                      "  Clearing Soft References>\n");
126
127             /* Can't expand the heap any more.  Try GC again but this time
128                clearing all soft references.  Note we succeed if we can
129                satisfy the request -- we may have been able to all along,
130                but with nothing spare.  We may thrash, but it's better
131                than throwing OOM */
132             largest = gc0_pmem(FALSE, TRUE);
133             if(n <= largest) {
134               state = gc;
135               break;
136             }
137
138             if(verbosegc)
139               jam_printf("<GC: completely out of heap space"
140                      " - throwing OutOfMemoryError>\n");
141
142             state = throw_oom;
143             unlockVMLock(heap_lock, self);
144             signalException(java_lang_OutOfMemoryError, NULL);
145             return NULL;
146             break;
147
148           case throw_oom:
149             /* Already throwing an OutOfMemoryError in some thread.  In
150                both cases, throw an already prepared OOM (no stacktrace).
151                Could have a * per-thread flag, so we try to throw a new
152                OOM in each thread, but if we're this low on memory I
153                doubt it'll make much difference.  */
154
155             if(verbosegc)
156               jam_printf("<GC: completely out of heap space"
157                      " - throwing prepared OutOfMemoryError>\n");
158
159             state = gc;
160             unlockVMLock(heap_lock, self);
161             setException(oom);
162             return NULL;
```

```
163             break;
164         }
165       }
166     }
167
168 got_it_pmem:
169     if(nvml_alloc) {
170       NVML_DIRECT("HEAPFREE", &(pheap->heapfree), sizeof(pheap->heapfree))
171     }
172
173     pheap->heapfree -= n;
174
175     if(have_remaining) {
176       if(nvml_alloc) {
177         NVML_DIRECT("FOUND", found, HEADER_SIZE * 2 + n)
178       }
179     }
180     else {
181       if(nvml_alloc) {
182         NVML_DIRECT("FOUND2", found, HEADER_SIZE + n)
183       }
184     }
185
186     found->header = n | ALLOC_BIT;
187
188     /* Found is a block pointer - if we unlock now, small window
189      * where new object ref is not held and will therefore be gc'ed.
190      * Setup ret_addr before unlocking to prevent this.
191      */
192
193     ret_addr = ((char*)found)+HEADER_SIZE;
194     memset(ret_addr, 0, n-HEADER_SIZE);
195     unlockVMLock(heap_lock, self);
196
197     return ret_addr;
198 }
```

A.2.7 Non-Heap Objects Persistence

We need to ensure that all data structures that are referenced by heap objects, such as class definitions and compiled methods, are persisted and can be retrieved in future executions. However, many of these structures are stored outside of the heap allocation space in the original JamVM implementation. In order to overcome this problem, we created a second persistent allocation space, using the same chunk-based management mechanism used for the heap.

Chunks managed in this memory region have the format defined in Listing A.18.

Listing A.18 – NVMChunk struct (jam.h)

```
typedef struct nvmChunk
{
 int allocBit;
 unsigned int chunkSize;
 struct nvmChunk *next;
} nvmChunk;
```

NVMChunk member allocBit indicates if the chunk is being used, chunkSize is the size of the chunk in bytes and next is a pointer to the next chunk on the list. As we are replacing the standard malloc function an alignment is necessary due to JamVM pointer masks.

A new set of allocation functions were created to handle data structures in this non-heap persistent space, as listed below.

- sysMalloc_persistent

  This function, as depicted on Listing A.19, searches for the first free chunk that can hold the specified size, allocates that chunk, and returns its address.

Listing A.19 – sysMalloc_persistent function (alloc.c)

```
void *sysMalloc_persistent(unsigned int size){
  if (persistent){
    uint n = size < sizeof(void*) ? sizeof(void*) : size;
    uint have_remaining = FALSE;
    void *ret_addr = NULL;
    nvmChunk *rem = NULL;
    nvmChunk *found = NULL;
    int err;
    int shift = 0;

    unsigned int len = 0;
    if(nvml_alloc) {
      NVML_DIRECT("NVMCHUNKPP", &pheap->nvmChunkpp, sizeof(nvmChunk*))
    }
    while (*(pheap->nvmChunkpp)){

      /*  search unallocated chunk    */
      if ((*(pheap->nvmChunkpp))->allocBit == 1) {
        pheap->nvmChunkpp = &(*(pheap->nvmChunkpp))->next;
        continue;
      }
      len = (*(pheap->nvmChunkpp))->chunkSize;

      if(len > n) {
        found = *(pheap->nvmChunkpp);
        /*  check if remaining space can hold a chunk   */
```

```
27          if((int)(len - (nvmHeaderSize + n)) >= minSize) {
28            /*  ptr + header  + content = OK  */
29            rem = (nvmChunk*)((char*)found + nvmHeaderSize + n);
30
31            have_remaining = TRUE;
32            if (shift = (unsigned int)rem & 0x3){
33              shift = (0x4 - shift);
34              memset(rem, 0 , shift);
35              rem = ((char*)rem + shift);
36            }
37            rem->allocBit = 0;
38            rem->chunkSize = len - (shift + nvmHeaderSize + n);
39            rem->next = found->next;
40          }
41          ret_addr = ((void*)found + nvmHeaderSize);
42          break;
43        }
44
45        if(len == n) {
46          found = *(pheap->nvmChunkpp);
47          ret_addr = ((void*)found + nvmHeaderSize);
48          break;
49        }
50        pheap->nvmChunkpp = &(*(pheap->nvmChunkpp))->next;
51      }
52
53      if (found == NULL) {
54        // we couldn't find any chunk with the desired size
55        // since we don't have GC yet, we'll have to exit the JVM
56        jam_fprintf(stderr, "We're out of chunks and don't have GC for
              sysMalloc_persistent.\n");
57        exitVM(1);
58      }
59
60      if(have_remaining) {
61        if(nvml_alloc) {
62          NVML_DIRECT("FOUND3", found, nvmHeaderSize * 2  + n)
63        }
64      }
65      else {
66        if(nvml_alloc) {
67          NVML_DIRECT("FOUND4", found, nvmHeaderSize + n)
68        }
69      }
70
71      found->allocBit = 1;
72      found->chunkSize = n;
```

```
73
74      if(have_remaining)
75        found->next = rem;
76
77      if(nvml_alloc) {
78        NVML_DIRECT("NVMFREESPACE", &(pheap->nvmFreeSpace), sizeof(pheap->
              nvmFreeSpace))
79      }
80
81      if(have_remaining)
82        pheap->nvmFreeSpace = pheap->nvmFreeSpace - nvmHeaderSize - shift
              - sizeof(nvmChunk);
83      else
84        pheap->nvmFreeSpace = pheap->nvmFreeSpace - found->chunkSize -
              sizeof(nvmChunk);
85
86      if (nvml_alloc) {
87        if ((void*)found >= (void*)(pheap->nvm + pheap->nvmCurrentSize)) {
88          printf("NVM_LIMIT exceeded %p >= %p\n", (void*)found, (void*)
                pheap->nvm_limit);
89        }
90      }
91
92      /* clear chunk */
93      memset(ret_addr, 0, found->chunkSize);
94
95      return ret_addr;
96    }else
97      return sysMalloc(size);
98  }
```

- sysFree_persistent

  This function (Listing A.20) sets the `allocBit` of a given chunk to zero, indicating its now free, and erases its content.

Listing A.20 – sysFree_persistent function (alloc.c)

```
1  void sysFree_persistent(void* addr) {
2      if(persistent) {
3          int err;
4          unsigned long ptr = (unsigned long) addr;
5          /*  chunk = ptr - header */
6          nvmChunk *toFree = (nvmChunk*)(ptr-nvmHeaderSize);
7          if(nvml_alloc) {
8              NVML_DIRECT("NVMFREESPACE", &(pheap->nvmFreeSpace), sizeof(
                   pheap->nvmFreeSpace))
9          }
```

```
10          pheap->nvmFreeSpace = pheap->nvmFreeSpace + toFree->chunkSize;
11          if(nvml_alloc) {
12              NVML_DIRECT("TOFREE", toFree, toFree->chunkSize +
                    nvmHeaderSize)
13          }
14          toFree->allocBit = 0;
15      memset((void*)ptr, 0, toFree->chunkSize);
16      } else {
17          sysFree(addr);
18      }
19  }
```

- sysRealloc_persistent

  This function (Listing A.21) is similar to standard `realloc` function: it copies the content to a new chunk with the given size, erases the old data and set the old chunk as free.

Listing A.21 – sysRealloc_persistent function (alloc.c)

```
1  void *sysRealloc_persistent(void *addr, unsigned int size) {
2      void *mem;
3      if (persistent) {
4          /*  chunk = ptr - header */
5          nvmChunk *toRealloc = (addr-nvmHeaderSize);
6          mem = sysMalloc_persistent(size);
7      if (addr != NULL) {
8          if ((toRealloc->chunkSize <= size)) {
9              memcpy(mem, ((void*)toRealloc+nvmHeaderSize), toRealloc->
                   chunkSize);
10         } else {
11             memcpy(mem, ((void*)toRealloc+nvmHeaderSize), size);
12         }
13         sysFree_persistent(addr);
14     }
15     return mem;
16     } else {
17         return sysRealloc(addr, size);
18     }
19 }
```

A.2.8 Hash Tables Persistence

JamVM stores several internals hash tables containing metadata. In JaphaVM, these hash tables are contained by the `PHeap` data structure, and the `gcMemMalloc` function (listed on A.22) returns pointers to where each hash table is stored in the non-heap area.

The Hash table initialization code was changed to avoid resetting all entries during JVM startup.

The DLL hash table is not stored, and thus needs to be recreated at each JVM execution.

Listing A.22 – gcMemMalloc function (alloc.c)

```
1  void *gcMemMalloc(int n, char* name, int create_file) {
2      uintptr_t size = n + sizeof(uintptr_t);
3      uintptr_t *mem;
4      int fd;
5
6      if (persistent && create_file) {
7          if(strcmp(name, HT_NAME_UTF8) == 0) {
8              return pheap->utf8_ht;
9          } else
10         if(strcmp(name, HT_NAME_BOOT) == 0) {
11             return pheap->bootCl_ht;
12         } else
13         if(strcmp(name, HT_NAME_BOOTPKG) == 0) {
14             return pheap->bootPck_ht;
15         } else
16         if(strcmp(name, HT_NAME_STRING) == 0) {
17             return pheap->string_ht;
18         } else
19         if(strcmp(name, HT_NAME_CLASS) == 0) {
20             return pheap->classes_ht;
21         } else
22         if(strcmp(name, HT_NAME_MONITOR) == 0) {
23             return pheap->monitor_ht;
24         } else
25         if(strcmp(name, HT_NAME_MONITOR) == 0) {
26             return pheap->monitor_ht;
27         } else
28         if(strcmp(name, HT_NAME_ZIP) == 0) {
29             return pheap->zip_ht;
30        }
31
32     unsigned long buffer[1];
33     size = size + sizeof(unsigned long);
34     if(access( name, F_OK ) != -1 ) {
35       fd = open (name, O_RDWR | O_APPEND , S_IRUSR | S_IWUSR);
36       read(fd, &buffer, sizeof(unsigned long)+sizeof(uintptr_t));
37       mem = (uintptr_t*)mmap((void*)buffer[0],buffer[1], PROT_READ|PROT_WRITE
             , MAP_SHARED, fd, 0);
38       *mem++ = (unsigned long)mem;
39     } else {
40       fd = open (name, O_RDWR | O_CREAT , S_IRUSR | S_IWUSR);
41       lseek (fd, size-1, SEEK_SET);
42       write(fd,"",1);
43       mem = (uintptr_t*)mmap(0, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd,
```

```
            0);
44        *mem++ = (unsigned long)mem;
45      }
46      msync(mem, size, MS_SYNC);
47      } else {
48          mem = mmap(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANON, -1,
                0);
49      }
50
51      if(mem == MAP_FAILED) {
52          jam_fprintf(stderr, "Mmap failed - aborting VM...\n");
53          exitVM(1);
54      }
55
56      *mem++ = size;
57      return mem;
58  }
```

A.2.9 Context Recovery

JamVM uses several static values that are set during *load*, *link* or *init* of classes. As in persistent mode it will only pass through these steps once (on first execution) a new mechanism was made to set this values in every subsequent execution.

A struct called `OPC` (Orthogonal Persistence Context), shown previously on Listing A.12, contains the most up to date value of the current execution and is updated at the end of each NVML transaction.

The `OPC` member `first_ex` is a flag indicating if the current execution is the first one.

A.2.10 Java Opcodes

The implementation of all Java opcodes that change heap state must store/access data inside the Memory Pool, and ensure that all modifications happen in transactional context. Listings A.23 — A.35 describe all changes made.

- ARRAY_STORE MACRO

Listing A.23 – ARRAY STORE macro (interp/engine/interp.c)

```
1  #define ARRAY_STORE(TYPE)                                \
2  {                                                        \
3      if(persistent) {                                     \
4              BEGIN_TX("ARRAY_STORE")                      \
5      }                                                    \
6      int val = ARRAY_STORE_VAL;                           \
7      int idx = ARRAY_STORE_IDX;                           \
```

```
8     Object *array = (Object *)*--ostack;       \
9                                                \
10    NULL_POINTER_CHECK(array);                 \
11    ARRAY_BOUNDS_CHECK(array, idx);            \
12    if(persistent) {                           \
13            NVML_DIRECT("ARRAY_STORE",         \
14            &(ARRAY_DATA(array, TYPE)[idx]),   \
15            sizeof(val))                       \
16    }                                          \
17    ARRAY_DATA(array, TYPE)[idx] = val;        \
18    if(persistent) {                           \
19            END_TX("ARRAY_STORE")              \
20    }                                          \
21    DISPATCH(0, 1);                            \
22 }
```

- MONITOR_ENTER OPCODE

Listing A.24 – MONITOR ENTER macro (interp/engine/interp.c)

```
1 DEF_OPC_210(OPC_MONITORENTER, {
2   Object *obj = (Object *)*--ostack;
3   NULL_POINTER_CHECK(obj);
4   if(persistent) {
5     BEGIN_TX("MONITORENTER")
6     NVML_DIRECT("ENTEROBJ", obj, sizeof(Object));
7   }
8   objectLock(obj);
9   DISPATCH(0, 1);
10 })
```

- MONITOR_EXIT OPCODE

Listing A.25 – MONITOR EXIT macro (interp/engine/interp.c)

```
1 DEF_OPC_210(OPC_MONITOREXIT, {
2   Object *obj = (Object *)*--ostack;
3   NULL_POINTER_CHECK(obj);
4   objectUnlock(obj);
5   if(persistent) {
6     END_TX("MONITOREXIT")
7   }
8   DISPATCH(0, 1);
9 })
```

- AASTORE OPCODE

Listing A.26 – AASTORE opcode (interp/engine/interp.c)

```
DEF_OPC_012(OPC_AASTORE, {
  if(persistent) {
    BEGIN_TX("AASTORE")
  }
  Object *obj = (Object*)ARRAY_STORE_VAL;
  int idx = ARRAY_STORE_IDX;
  Object *array = (Object *)*--ostack;

  NULL_POINTER_CHECK(array);
  ARRAY_BOUNDS_CHECK(array, idx);

  if((obj != NULL) && !arrayStoreCheck(array->class, obj->class))
    THROW_EXCEPTION(java_lang_ArrayStoreException, NULL);

  if(persistent) {
    NVML_DIRECT("AASTORE", &(ARRAY_DATA(array, Object*)[idx]), sizeof(
        obj))
  }
  ARRAY_DATA(array, Object*)[idx] = obj;
  if(persistent) {
    END_TX("AASTORE")
  }
  DISPATCH(0, 1);
})
```

- LASTORE and DASTORE OPCODES

Listing A.27 – LASTORE and DASTORE opcodes (interp/engine/interp.c)

```
DEF_OPC_012_2(
    OPC_LASTORE,
    OPC_DASTORE, {
  if(persistent) {
    BEGIN_TX("LASTORE - DASTORE")
  }
  int idx = ostack[-3];
  Object *array = (Object *)ostack[-4];

  if(persistent) {
    NVML_DIRECT("L&D ASTORE", &(ARRAY_DATA(array, u8)[idx]), sizeof(u8))
  }
  ostack -= 4;
  NULL_POINTER_CHECK(array);
  ARRAY_BOUNDS_CHECK(array, idx);

  ARRAY_DATA(array, u8)[idx] = *(u8*)&ostack[2];
```

```
18   if(persistent) {
19     END_TX("LASTORE - DASTORE")
20   }
21   DISPATCH(0, 1);
22 })
```

- NEW_QUICK OPCODE

Listing A.28 – NEW QUICK opcode (interp/engine/interp.c)

```
1 DEF_OPC_210(OPC_NEW_QUICK, {
2   if(persistent) {
3     BEGIN_TX("NEW_QUICK")
4     nvml_alloc = TRUE;
5   }
6   Class *class = RESOLVED_CLASS(pc);
7   Object *obj;
8
9   frame->last_pc = pc;
10  if((obj = allocObject(class)) == NULL)
11    goto throwException;
12
13  if(persistent) {
14    END_TX("NEW_QUICK")
15    nvml_alloc = FALSE;
16  }
17  PUSH_0((uintptr_t)obj, 3);
18 })
```

- NEWARRAY OPCODE

Listing A.29 – NEWARRAY opcode (interp/engine/interp.c)

```
1 DEF_OPC_210(OPC_NEWARRAY, {
2   if(persistent) {
3     BEGIN_TX("NEWARRAY")
4     nvml_alloc = TRUE;
5   }
6   int type = ARRAY_TYPE(pc);
7   int count = *--ostack;
8   Object *obj;
9
10  frame->last_pc = pc;
11  if((obj = allocTypeArray(type, count)) == NULL)
12    goto throwException;
13
14  if(persistent) {
15    END_TX("NEWARRAY")
```

```
16      nvml_alloc = FALSE;
17    }
18    PUSH_0((uintptr_t)obj, 2);
19  })
```

- ANEWARRAY_QUICK OPCODE

Listing A.30 – ANEWARRAY QUICK opcode (interp/engine/interp.c)

```
1  DEF_OPC_210(OPC_ANEWARRAY_QUICK, {
2    if(persistent) {
3      BEGIN_TX("ANEWARRAY_QUICK")
4      nvml_alloc = TRUE;
5    }
6    Class *class = RESOLVED_CLASS(pc);
7    char *name = CLASS_CB(class)->name;
8    int count = *--ostack;
9    Class *array_class;
10   char *ac_name;
11   Object *obj;
12
13   frame->last_pc = pc;
14
15   if(count < 0) {
16     signalException(java_lang_NegativeArraySizeException, NULL);
17     goto throwException;
18   }
19
20   ac_name = sysMalloc(strlen(name) + 4);
21
22   if(name[0] == '[')
23     strcat(strcpy(ac_name, "["), name);
24   else
25     strcat(strcat(strcpy(ac_name, "[L"), name), ";");
26
27   array_class = findArrayClassFromClass(ac_name, mb->class);
28   free(ac_name);
29
30   if(exceptionOccurred0(ee))
31     goto throwException;
32
33   if((obj = allocArray(array_class, count, sizeof(Object*))) == NULL)
34     goto throwException;
35
36   if(persistent) {
37     END_TX("ANEWARRAY_QUICK")
38     nvml_alloc = FALSE;
```

```
39   }
40   PUSH_0((uintptr_t)obj, 3);
41 })
```

- MULTIANEWARRAY_QUICK OPCODE

Listing A.31 – MULTIANEWARRAY QUICK opcode (interp/engine/interp.c)

```
1  DEF_OPC_210(OPC_MULTIANEWARRAY_QUICK, ({
2    if(persistent) {
3      BEGIN_TX("MULTIANEWARRAY_QUICK")
4      nvml_alloc = TRUE;
5    }
6    Class *class = RESOLVED_CLASS(pc);
7    int i, dim = MULTI_ARRAY_DIM(pc);
8    Object *obj;
9
10   ostack -= dim;
11   frame->last_pc = pc;
12
13   for(i = 0; i < dim; i++)
14     if((intptr_t)ostack[i] < 0) {
15       signalException(java_lang_NegativeArraySizeException, NULL);
16       goto throwException;
17     }
18
19   if((obj = allocMultiArray(class, dim, (intptr_t *)ostack)) == NULL)
20     goto throwException;
21
22   if(persistent) {
23     END_TX("MULTIANEWARRAY_QUICK")
24     nvml_alloc = FALSE;
25   }
26   PUSH_0((uintptr_t)obj, 4);
27 });)
```

- PUTSTATIC_QUICK OPCODE

Listing A.32 – PUTSTATIC QUICK opcode (interp/engine/interp.c)

```
1  DEF_OPC(OPC_PUTSTATIC_QUICK##suffix, level,            \
2    if(persistent) {                              \
3       BEGIN_TX("PUTSTATIC_QUICK")                      \
4       NVML_DIRECT("PUTSTATICQUICK",                    \
5       RESOLVED_FIELD(pc), sizeof(FieldBlock));         \
6       END_TX("PUTSTATIC_QUICK")                        \
7    }                                          \
8    POP_##level(*(type*)                               \
```

```
9       (RESOLVED_FIELD(pc)->u.static_value.data), 3);   \
10 )
```

- PUTSTATIC2_QUICK OPCODE

Listing A.33 – PUTSTATIC2 QUICK opcode (interp/engine/interp.c)

```
1 DEF_OPC_012(OPC_PUTSTATIC2_QUICK, {
2   if(persistent) {
3     BEGIN_TX("PUTSTATIC2_QUICK")
4   }
5   FieldBlock *fb = RESOLVED_FIELD(pc);
6   if(persistent) {
7     NVML_DIRECT("PUTSTATIC2_QUICK", fb, sizeof(FieldBlock))
8     END_TX("PUTSTATIC2_QUICK")
9   }
10  POP_LONG(fb->u.static_value.l, 3);
11 })
```

- PUTFIELD_QUICK OPCODE

Listing A.34 – PUTFIELD QUICK opcode (interp/engine/interp.c)

```
1 DEF_OPC_012(OPC_PUTFIELD_QUICK##suffix, {                \
2     if(persistent) {                                     \
3         BEGIN_TX("PUTFIELD_QUICK")                       \
4     }                                                    \
5     Object *obj = (Object *)ostack[-2];                  \
6                                                          \
7     ostack -= 2;                                         \
8     NULL_POINTER_CHECK(obj);                             \
9     if(persistent) {                                     \
10        NVML_DIRECT("PUTFIELD_QUICK",&(INST_DATA(obj,    \
11        type, SINGLE_INDEX(pc))),sizeof(type))           \
12    }                                                    \
13    INST_DATA(obj, type, SINGLE_INDEX(pc)) = ostack[1]; \
14    if(persistent) {                                     \
15        END_TX("PUTFIELD_QUICK")                         \
16    }                                                    \
17    DISPATCH(0, 3);                                      \
18 })
```

- PUTFIELD2_QUICK OPCODE

Listing A.35 – PUTFIELD2 QUICK opcode (interp/engine/interp.c)

```
1 DEF_OPC_012(OPC_PUTFIELD2_QUICK, {
2   if(persistent) {
```

```
3     BEGIN_TX("PUTFIELD2_QUICK")
4   }
5   Object *obj = (Object *)ostack[-3];
6
7   ostack -= 3;
8   NULL_POINTER_CHECK(obj);
9   if(persistent) {
10    NVML_DIRECT("PUTFIELD2_QUICK",&(INST_DATA(obj, u8, SINGLE_INDEX(pc))
          ),sizeof(u8))
11  }
12  INST_DATA(obj, u8, SINGLE_INDEX(pc)) = *(u8*)&ostack[1];
13  if(persistent) {
14    END_TX("PUTFIELD2_QUICK")
15  }
16  DISPATCH(0, 3);
17 })
```

A.2.11 Garbage Collection

As Garbage Collection (GC) modified the persistent heap state, it must happen in the context of a NVML transaction. We have created new versions of the GC function, which is used when the persistent heap is enabled: `gc0_pmem` (Listing A.36).

Listing A.36 – Function gc0_pmem (alloc.c)

```
1 unsigned long gc0_pmem(int mark_soft_refs, int compact) {
2     Thread *self = threadSelf();
3     uintptr_t largest;
4
5     /* Override compact if compaction has been specified
6        on the command line */
7     if(compact_override)
8         compact = compact_value;
9
10    /* Reset flags.  Will be set during GC if a thread needs
11       to be woken up */
12    notify_finaliser_thread = notify_reference_thread = FALSE;
13
14    /* Grab locks associated with the suspension blocked
15       regions.  This ensures all threads have suspended
16       or gone to sleep, and cannot modify a list or obtain
17       a reference after the reference scans */
18
19    /* Potential threads adding a newly created object */
20    lockVMLock(has_fnlzr_lock, self);
21
22    /* Held by the finaliser thread */
```

```
23  lockVMWaitLock(run_finaliser_lock, self);
24
25    /* Held by the reference handler thread */
26    lockVMWaitLock(reference_lock, self);
27
28    /* Stop the world */
29    disableSuspend(self);
30    suspendAllThreads(self);
31
32    if(verbosegc) {
33        struct timeval start;
34        float mark_time;
35        float scan_time;
36
37        getTime(&start);
38    BEGIN_TX("GC-VERBOSE");
39        doMark(self, mark_soft_refs);
40        mark_time = endTime(&start)/1000000.0;
41
42        getTime(&start);
43        largest = compact ? doCompact() : doSweep(self);
44    END_TX("GC-VERBOSE");
45        scan_time = endTime(&start)/1000000.0;
46
47        jam_printf("<GC: Mark took %f seconds, %s took %f seconds>\n",
48                            mark_time, compact ? "compact" : "scan", scan_time
                               );
49    } else {
50    BEGIN_TX("GC");
51        doMark(self, mark_soft_refs);
52        largest = compact ? doCompact() : doSweep(self);
53    END_TX("GC");
54    }
55
56    /* Restart the world */
57    resumeAllThreads(self);
58    enableSuspend(self);
59
60    /* Notify the finaliser thread if new finalisers
61       need to be ran */
62    if(notify_finaliser_thread)
63        notifyAllVMWaitLock(run_finaliser_lock, self);
64
65    /* Notify the reference thread if new references
66       have been enqueued */
67    if(notify_reference_thread)
68        notifyAllVMWaitLock(reference_lock, self);
```

```
69
70      /* Release the locks */
71      unlockVMLock(has_fnlzr_lock, self);
72      unlockVMWaitLock(reference_lock, self);
73      unlockVMWaitLock(run_finaliser_lock, self);
74
75      freeConservativeRoots();
76      freePendingFrees();
77
78      return largest;
79  }
```

The gc0_pmem function is called in two situations: a) 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; b) during an asynchronous GC, invoked by a specialized thread that triggers a GC periodically when the VM is idle. In the second case we need to start a new transaction, but can only do that if there are no concurrent transactions, to avoid isolation problems. In order to solve this problem, we have added a global transaction counter (updated every time that BEGIN_TX / END_TX are called, and check this counter before triggering an asynchronous GC (see Listing A.37).

Listing A.37 – Function gc1 (alloc.c)

```
1  void gc1() {
2      Thread *self;
3      disableSuspend(self = threadSelf());
4      lockVMLock(heap_lock, self);
5    // JAPHA modification to add async tx GC
6    if(verbosegc) jam_printf("<GC: Attempting async GC>, tx_count=%u\n",
         total_tx_count);
7    if (total_tx_count == 0) {    // only perform async GC if there is no
         ongoing NVML transaction
8      if(verbosegc) jam_printf("<GC: Will execute async GC>\n");
9      enableSuspend(self);
10     if (persistent) {
11       gc0_pmem(TRUE, FALSE);
12     } else {
13       gc0(TRUE, FALSE);
14     }
15   }
16   // end of JAPHA modification
17     unlockVMLock(heap_lock, self);
18 }
```

Finally, two data structures have to be persisted in order for the garbage collection process to work properly:

- Primary classes array

  Primary classes are allocated inside an array lazily, so to keep the data context the array had to be persisted. We added this array to `OPC->prim_classes` (see code listing A.12).

- Has finalizer list

  Objects that have a finalizer are stored inside a list, so to keep the context of objects that have finalizers the list had to be persisted. We added this to `OPC->has_finaliser_list` (see code listing A.12).

A.2.12 Handling External State: Console I/O

As described in section 4.9.5, 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 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 [39] determines that a static initializer is executed when the class is initialized, which happens one time 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 provide a way to express this.

We have taken the approach used previously for PJama and PEVM [9,51], exposing for developers an API to make classes and objects *resumable*, i.e., 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. In order to execute resume code at the class scope, the class must implement a static `void resume()` method and be registered by the `OPRuntime.addStaticListener()` class method.

We have implemented the `OPRuntime` class as part of GNU Classpath (see code listing A.38). In order to get the `resume()` methods to be executed whenever the JVM resumes execution, we have modified the JamVM to add a `resumeAllListeners` function, called just before the execution of the Java `main` method. See code listing A.39.

Finally, we applied this solution to reopen the console file descriptors, as shown in code listing A.40.

Listing A.38 – OPRuntime class (OPRuntime.java)

```
1 /**
2  * This class registers OPResumeListener objects for execution
3  * upon JVM re-initialization.
4  */
5 public class OPRuntime {
6
```

```
7   private static Set<OPResumeListener> listeners = null;
8
9   private static Set<Class<?>> staticListeners = null;
10
11  /**
12   * Adds a new listener.
13   */
14  public static void addListener(OPResumeListener listener) {
15    // initialize listeners set if necessary
16    if (listeners == null) {
17      listeners = new HashSet<OPResumeListener>();
18    }
19    // add listener to listeners set
20    listeners.add(listener);
21  }
22
23  /**
24   * Adds a new listener to a class that has a static resume() method.
25   */
26  public static void addStaticListener(Class<?> clazz) {
27    // initialize listeners set if necessary
28    if (staticListeners == null) {
29      staticListeners = new HashSet<Class<?>>();
30    }
31    // add listener to listeners set
32    staticListeners.add(clazz);
33  }
34
35  /**
36   * Method called by the JVM runtime to re-initialize OPResumeListener
        objects.
37   */
38  public static void resumeAllListeners() {
39    // resume all listeners
40    if (listeners != null) {
41      Iterator<OPResumeListener> it = listeners.iterator();
42      while(it.hasNext()) {
43        OPResumeListener listener = it.next();
44        listener.resume();
45      }
46    }
47
48    // resume all static listeners
49    if (staticListeners != null) {
50      Iterator<Class<?>> it = staticListeners.iterator();
51      while(it.hasNext()) {
52        Class<?> clazz = it.next();
```

```
53          Method method;
54          try {
55            method = clazz.getMethod("resume", null);
56            method.invoke(null, (Object[])null);
57          } catch (NoSuchMethodException e) {
58            e.printStackTrace();
59          } catch (SecurityException e) {
60            e.printStackTrace();
61          } catch (IllegalAccessException e) {
62            e.printStackTrace();
63          } catch (IllegalArgumentException e) {
64            e.printStackTrace();
65          } catch (InvocationTargetException e) {
66            e.printStackTrace();
67          }
68        }
69      }
70    }
71  }
```

Listing A.39 – resumeAllListeners() function (jam.c)

```
1  int resumeAllListeners(Object *system_loader)
2  {
3    Class *op_runtime = findClassFromClassLoader("javax.op.OPRuntime",
         system_loader);
4    Class *vm_channel = findClassFromClassLoader("gnu.java.nio.VMChannel",
         system_loader);
5
6      if(op_runtime != NULL)
7          initClass(op_runtime);
8
9      if(exceptionOccurred())
10         return FALSE;
11
12     MethodBlock *mb = lookupMethod(op_runtime, SYMBOL(resumeAllListeners),
13                                    SYMBOL(___V));
14
15     if(mb == NULL || !(mb->access_flags & ACC_STATIC))
16     {
17         signalException(java_lang_NoSuchMethodError, "resumeAllListeners");
18         return FALSE;
19     }
20
21     executeStaticMethod(op_runtime, mb, NULL);
22
23     if(vm_channel != NULL)
```

```
24        initClass(vm_channel);
25
26      if(exceptionOccurred())
27        return FALSE;
28
29      if((mb = findMethod(vm_channel, SYMBOL(class_init), SYMBOL(___V))) !=
            NULL)
30          executeStaticMethod(vm_channel, mb);
31
32      return TRUE;
33
34  }
```

Listing A.40 – FileChannelImpl resume implementation (FileChannelImpl.java)

```
1  ...
2  public static FileChannelImpl in;
3  public static FileChannelImpl out;
4  public static FileChannelImpl err;
5    //private static native void init();
6    static
7    {
8      // XXX NVM CHANGE - moved initialization block code to resume() method,
           which implements the OPResumeListener
9      resume();
10     OPRuntime.addStaticListener(FileChannelImpl.class);
11   }
12   public static void resume() {
13     if (Configuration.INIT_LOAD_LIBRARY)
14       {
15         System.loadLibrary("javanio");
16       }
17     FileChannelImpl ch = null;
18     try
19       {
20         ch = new FileChannelImpl(VMChannel.getStdin(), READ);
21       }
22     catch (IOException ioe)
23       {
24         throw new Error(ioe);
25       }
26     in = ch;
27     ch = null;
28     try
29       {
30         ch = new FileChannelImpl(VMChannel.getStdout(), WRITE);
31       }
```

```
32      catch (IOException ioe)
33        {
34          throw new Error(ioe);
35        }
36      out = ch;
37      ch = null;
38      try
39        {
40          ch = new FileChannelImpl(VMChannel.getStderr(), WRITE);
41        }
42      catch (IOException ioe)
43        {
44          throw new Error(ioe);
45        }
46      err = ch;
47    }
48  ...
```

A.2.13 JaphaVM Prototype Limitations

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;
- 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 vs. execution persistence — the current version supports only data persistence (see Sections 4.7.1, 4.7.2);
- Type evolution — JaphaVM currently does not support type evolution.

Despite these limitations, the current JaphaVM prototype can execute a wide range of complex programs, and functions as a vehicle for both performance and development complexity experiments and evaluation, as presented in chapter 5.