PLAVA: A PERSISTENT, LIGHTWEIGHT JAVA™ VIRTUAL MACHINE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Stephen Jeremy Tjasink
February 1999

Supervised by
S. Berman

Abstract

This thesis investigates the feasibility of running an orthogonally persistent version of Java™ on a "small" machine with limited memory available. Other orthogonally persistent implementations do exist but these are targetted at large data sets and powerful machines.

The modification of an existing lightweight Java Virtual Machine (JVM) in order to provide persistence is discussed. It highlights decisions that were made in the areas of memory management, object faulting and update tracking in order to implement these facilities efficiently without using undue amounts of memory. Also discussed are the problems of caching and of swapping reachable objects to the persistent store in order to make more memory available. It is possible to configure the machine to use different strategies for each of these mechanisms so that comparisions can be made. The machine is able to interface with PJSL, the persistent store being developed for PJama.

Applications such as Generalised Search Tree (GiST) indexing of data collections have been run on the machine, as well as traversals from the OO7 benchmark suite. The performance of different machine configurations has been measured.

Acknowledgements

Thanks to...

My supervisor Sonia Berman who first engendered my interest in persistent systems in lectures and then convinced me to try looking at the implementation side.

My fiancée Vicki who thankfully kept reminding me that finishing my thesis was more beneficial in the long term than watching cricket on television.

My parents for always giving me support even though I never quite managed to explain to them what all this is about.

Tony Printezis who was an indispensable help when I was trying to get PLaVa to work with PJSL.

Laurent Daynès who sent me his Java/PJama implementation of the OO7 benchmark, saving me many hours of trying to convert it myself.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Trends in the development of computer hardware are leading to computing devices becoming ever more powerful as well as ever smaller. On one end of the scale, we have increasingly powerful machines becoming available. On the other end of the scale, we have technology that was at the cutting edge a few years ago available in a much smaller and more efficient form. As a result, increasing numbers of small devices are becoming "intelligent" as small processors are embedded in them. To take advantage of this, we decided to implement a lightweight, persistent version of Java™ [1] which we've called PLaVa [2].

When Java was introduced by Sun Microsystems in 1995, many people believed it to be an ideal development environment for embedded systems and other small machines. Over the past couple of years however, Sun's standard Java distribution and its associated libraries have expanded tremendously in an attempt to provide more functionality in diverse application areas. This has left players in the small systems arena with a problem in the face of bloated Java implementations. (Sun has recently released an EmbeddedJava™ (EJava) specification [3] for such systems.)

Most existing work in the area of persistent Java is following a similar trend, focussing on providing efficiency in the face of large amounts of data encountered [4, 5, 6], e.g. in GIS and high-end business applications. Significant progress has been made in this area by e.g., the PJama [4] and JavaSPIN [7] projects. This thesis attempts to “plug the gap” by focussing on allowing persistent applications to run in an environment with restricted resources.

We use PJSL [8] as one of the persistent stores on which PLaVa programs can run. This demonstrates its suitability for use with Virtual Machines other than PJama [4] for which it was designed, as well as PLaVa’s adaptability to different store architectures. Java is developing very quickly and in the interest of helping persistent Java mature with it (and possibly move towards defining a persistent Java standard), alternative implementations need to be developed so that experience can be gained and exploited.

1.2 Background

Persistent programming languages [9] have attempted to bring together programming languages and database systems. When loading and storing complex data with a traditional programming language, the programmer must expend significant effort (and possibly introduce many errors) either to store the data in files or to convert the data into a format required by a database system. Using only a traditional database system can be insufficient for certain applications due to the restricted capabilities of database query languages.

The persistent programming approach seeks to eliminate both these problems by transparently combining the expressive power of a programming language with the storage capabilities of a database system. This means that data is stored in a persistent store, but the programmer sees this as part of the heap and does not need to concern himself with the processes of storage and retrieval. Such an approach has many apparent advantages, ultimately resulting in greater productivity, increased safety and better performance.

It would seem that this approach would be able to draw from expertise in both the database and programming language areas, yet persistent programming languages still fall short in supplying features such as recoverability, scalability and schema evolution (These and other shortcomings are discussed in [9] and [10]). Current work on features such as these leads to large, complex systems that are not suitable for platforms with limited resources.

Since the early days of persistent programming languages when languages such as PS-Algol [11] and LOOM [12] were available, up to current research in which persistent versions of Java [4, 7] are used, researchers have relied heavily on adding persistence to existing languages. In some cases existing languages have been extended significantly to produce a new language (e.g. C++ [13] extended to form E [14]) and a small number of new, purpose-designed languages have been created (e.g. Napier88 [15], Fibonacci [16]). The attraction of using an existing language is the probability that it will find a wider user base. The unfortunate side-effect is that existing languages often cannot provide easy ways to achieve required persistence functionality.

Java has emerged as a language that readily lends itself to orthogonal persistence. It is strongly typed which provides valuable safety. It is interpreted and does not have explicit pointers, which makes adding persistence easy and neat. All data in Java must be included in an object, which provides a convenient minimum granularity for persistence.

1.3 Approach

The approach that we took to providing a lightweight persistent Java was to modify the virtual machine and to keep the language itself untouched. This allows the vast library of existing Java code to run on the system without any modification and also allows applications to be compiled with standard distribution Java compilers. Only a few lines of additional Java code are required to load or save persistent roots, resulting in a high degree of transparency.

Implementation is kept as independent of underlying hardware as possible, to keep the system portable. To help in this respect, object faulting is used to bring objects into memory without any reliance on specific hardware. Software checks are used to distinguish between disk and memory addresses. Many independently configurable parameters are provided to tune the virtual machine and the effects of a number of these parameters are tested.

In order to minimise memory usage, both persistent and transient objects are stored in the same heap instead of maintaining a separate cache for persistent objects. To accomodate this, garbage collection is extended to include eviction of reachable objects from the heap when necessary. This leads to object swapping which allows us to keep objects on the store temporarily.

1.4 Organisation of this Thesis

The rest of this thesis is arranged as follows:

Chapter 2 gives an overview of relevant literature that was consulted during this project. Many areas are covered, including language design, persistence implementations, optimisations and store technologies.

Chapter 3 covers the design and implementation of LEAN, a portable lightweight Java virtual machine on which PLaVa is based.

Chapter 4 discusses the design at the language level. The language syntax has remained untouched and this short chapter concentrates on describing the Java interface to the store as well as pointing out slight changes in the semantics that are introduced as a result of persistence.

Chapter 5 presents the first version of PLaVa ($PLaVa_1$) which was heavily dependent on the design of the machine discussed in chapter 3. Direct addressing is used and the heap needs to be large enough for all currently reachable objects to be resident at every point in the program's execution.

Chapter 6 presents the second version of PLaVa ($PLaVa_2$) which provides increased functionality over $PLaVa_1$. The most important of these improvements is the introduction of indirect addressing which allows objects to be swapped to the store.

Chapter 7 evaluates the work that has been done and presents some test cases. These include purpose-made tests used to compare different machine configurations as well as an adapted implementation of the OO7 benchmark suite [17].

Chapter 8 concludes, giving a summary of achievements and results.

Chapter 2

Background

2.1 Introduction

2.1.1 Building a Persistent System

In order to create a persistent language such as persistent Java, there are three main research areas to explore, corresponding to different levels of the design and implementation. At the lowest level is the persistent store itself; above this are the mechanisms for providing persistence during execution. These include provision for objects to be faulted in from the store and for writing dirty objects back to the store as well as cache management issues.

The top level is that of the language itself. If we use the approach of adding persistence to an existing programming language, alterations may have to be made to this language. Even if the syntax does not change, the addition of persistence might affect the language semantics.

2.1.2 Chapter Outline

This chapter gives an account of existing work that has been done in each of the areas mentioned above. It begins with an introduction to persistent language design and to Java, before presenting some existing persistent languages. Thereafter persistence mechanisms are examined, along with a number of optimisation techniques and alternative engineering strategies. This is followed by considering the bottom level of a persistent system architecture, by briefly outlining a few persistent stores.

The conclusion includes some experiences gained by the PJama team with early versions of their persistent Java system, and a review of current work.

2.2 Persistent Programming Language Design

Many researchers believe that when designing a persistent language, the implementer should strive for *orthogonal persistence*. In order to achieve this goal, a language must adhere to the following three principles of persistence:[9]

- The form of a program is the same whether it is manipulating persistent or transient data. (*The Principle of Persistence Independence*)
- Data of all types should be allowed to persist. (*The Principle of Data Type Orthogonality*)
- There should be a well defined system for identifying which data will become persistent. (*The Principle of Persistence Identification*)

When one designs a persistent system, there are four basic approaches [9] that can be taken. Three of these involve combining existing designs with new ones.

The first of the approaches is to *combine existing systems*. This requires interfacing an existing database system with an existing language, which can be problematic if the designs of the two separate systems are not amenable to this marriage. It will however allow existing data to be used via an existing programming language.

The second approach is to *extend an existing system*, either starting with a database system and adding a new language to it or starting with an existing language and adding custom-built persistence to it. Many languages that have persistence added to them in this fashion cannot achieve orthogonality as they were not designed to support persistence.

A number of systems have been developed using an *integrated design*, which is the third approach. This approach requires a new design for both the language and the database system and can draw on expertise from both these research areas.

The fourth approach differs from the other three in that it provides persistence by allowing applications to run in a *persistent world*. In this case, all programming languages used in this environment (e.g. a persistent operating system or special hardware that supports persistence) will automatically achieve persistence.

A large number of systems implement persistence by reachability. At least one *persistent root* is required for this. This root persists and all objects that are transitively reachable via references from it automatically persist as well. Objects will be loaded from the persistent store when they are needed and the system will keep track of which objects have been updated so that they can be written back to the store. Various mechanisms that can be used to achieve this are described later in this chapter.

2.3 Introduction to Java

Java is a modern, object-oriented programming language [18, 1]. It is loosely based on C++ [13] but does not include the commonly confusing features and constructs that have made writing reliable C++ applications so difficult. Java programs do not use explicit pointers and the heap is implicitly managed by a garbage collector; thus the programmer never has to worry about memory allocation and de-allocation. A built-in thread library is provided to help with the development of complex applications.

There is better handling of privacy and security in Java than in C++. Exception handling mechanisms are also provided in the language. Java allows only single inheritance and special classes called interface classes can be used to define behaviour, similarly to abstract super-classes in C++.

When Java source code is compiled, a Java *class file* is produced. This class file contains the compiled Java code as Java *bytecode* which is the "machine code" of the Java Virtual Machine (JVM) on which it is run. Because this standard bytecode is used, compiled Java classes can be run on any platform that has a JVM implemented

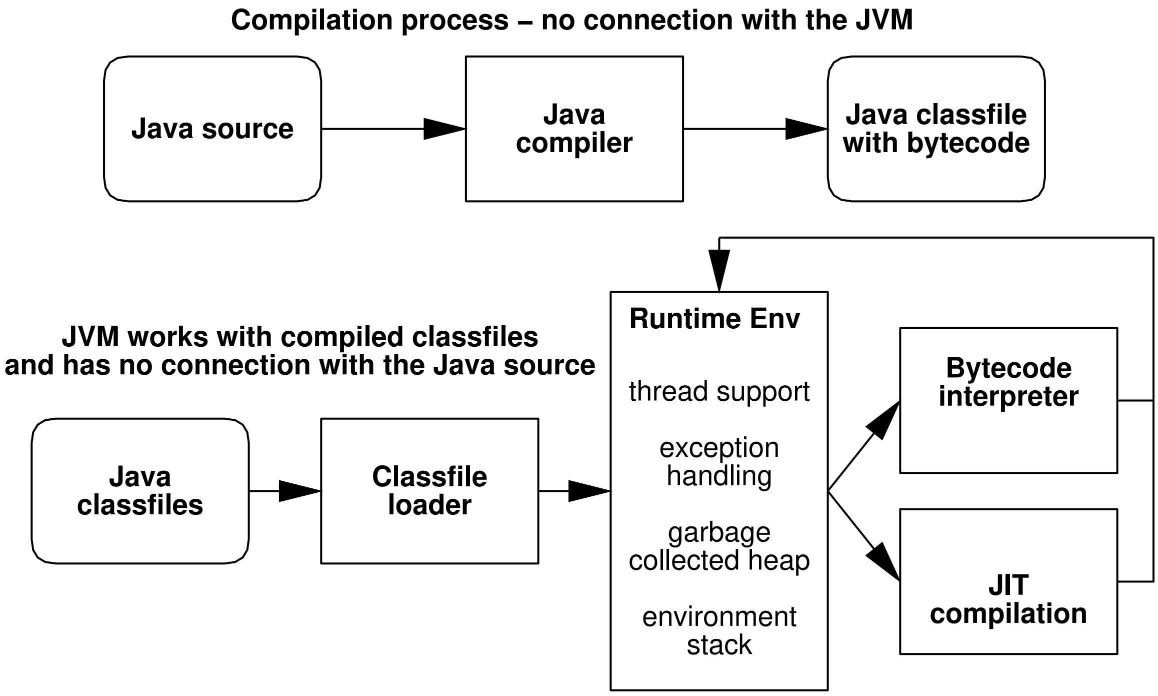

Figure 1: Depiction of the Java compilation and execution processes

for it.

2.4 Designing a Persistent Java

This section gives a quick overview of a number of existing persistent, object-oriented languages with the emphasis on how persistence is incorporated at the language level. Systems that incorporate persistence into Java are discussed first.

2.4.1 Persistent Java Language Design Issues

In [19] Moss and Hosking present a discussion of most of the design decisions that must be made when adding persistence to Java. Their analysis is conducted in two parts. The first deals with decisions that affect *orthogonality* and the second with those that affect *transparency* (i.e. *persistence independence*).

The first issue concerning orthogonality is whether or not to allow code to persist.

If code can persist, then the system becomes more self-contained. Not allowing code to persist is a simpler option, but for safety one will need to check that code (loaded from outside the store) has not been changed since the objects on which it acts were added to the store.

Allowing execution state to persist presents more barriers to orthogonality. Some aspects of program state such as open files may not be sensible to save.

The choice of a transaction model can affect performance. Offering complex transaction control will also mean that more changes will need to be made to existing virtual machines in order to support the transactions.

The definition of transparency usually relates to the source language. However, Java is unusual in that the specification includes not only the language but also the Java bytecode definition.

In order to maintain a high degree of transparency, the source language should not be changed. To avoid changing the language, one can extend other parts of the system or provide new libraries to support persistence. Regardless of the source language, if the output is standard Java bytecode then the object language remains portable. A bytecode post-processor could be used to add Java bytecode instructions into the compiled code to check for object residency, mark objects as changed, etc.

Obviously any approach must somehow change the implementation, but it may be possible to do so without actually changing the standard interpreter. It is likely that such an approach will have limitations, e.g. it will be hard to provide orthogonality.

2.4.2 PJama$_0$

The design of an orthogonally persistent Java system is given in [4]. It has the following goals:

1. No loss of safety when using persistence.
2. No impact on the performance of programs that do not use persistence. Minimum performance penalties and, if possible, performance gains for those that do.

3. Minimal changes to the language.
4. Simple use of persistence for programmers that can be satisfied by a default transaction model.
5. More precise transaction models for who that require it.

A `PJavaStore` object is the root of every store. To register an object as a root, the method `registerPRoot(aName, anObject)` is called to allow `anObject` to persist, and consequently everything reachable from it will persist as well. Objects remain persistent for as long as they are accessible from `PJavaStore` via at least one registered root.

When an object `c`, an instance of class `C`, is made to persist then all information that is needed by `c` must be made persistent. This includes the class `C`, `C`'s superclasses, all the objects reachable from `c`, all their classes and all the objects reachable via static variables of any of the classes involved.

Stabilisation is the process of making sure that data is written into the store. After a successful stabilise, data concerned will be available even if a software failure follows. A call of `PJavaStore`'s `stabilizeAll()` method will achieve a global stabilisation. The default transactional behaviour of PJama is an implicit `stabilizeAll` call at the end of each successfully terminated program. An uncaught exception has the same effect as a transaction abort.

In Java, a class's static variables are initialised when it is loaded. In PJama there are two possible interpretations of this:

- *persistently static* variables that are initialised once and always have their current value stored (the default)
- *transiently static* variables that are initialised every time that the class is loaded from the stable store.

An internal as well as an external transaction API is provided. The external API provided by instances of `TransactionShell` and its specialisations should satisfy most application programmers. Programmers who wish to define new transactional models by defining their own specialisations of `TransactionShell` can use the internal API to do so.

Since PJama$_0$ uses a modified virtual machine, it goes against Java's *run anywhere* guarantee. Other designs for persistent versions of Java do attempt to implement persistence entirely with the class library, but these are unlikely to achieve the desired level of orthogonality.

2.4.3 Early Use of PJama

[20] reports on experiences with the prototype persistent Java system PJama$_0$ and [21] gives an update on PJama's development.

Initially there was intent to allow access to multiple stores via multiple instances of the `PJavaStore` class but this idea has been abandoned due to the way that classes are loaded in Java.

The chosen model for PJama has been defined to be that of a suspended virtual machine. This means that a number executions against data on the store are seen as the same application execution.

If a class becomes unreachable and is unloaded, it will have to be loaded again if it is subsequently needed. This will cause its static variables to be re-initialised which might be undesirable behaviour. To combat this, a class is only considered unreachable if its class loader is unreachable.

Although an extensible transaction model has been planned for PJama, it has not yet been available with any public release of the system. The current system considers successful termination to include an implicit stabilise. Abnormal termination does not include this stabilise, thus resulting in an abort. No commercially available Java systems that support transactions allow an abort to undo changes made to transient objects. Java's existing concurrency mechanisms can be used to provide functionality more complex than simple ACID transactions.

It was originally argued that since native code posed risks to data consistency, it should not be allowed in PJama applications. However, native code is often useful if not essential and native code implementers must make sure that their code follows compatibility guidelines. There was no provision in PJama$_0$ to support persistence of native state and this thorny problem has since demanded some attention.

Problems arise with objects that store *external state* such as operating system file descriptors. It was thought that the `transient` modifier could be used to specify fields that needed to be re-initialised because of their reliance on native state, but the latest version of the Java Language Specification defines `transient` for a specific use in Java Object Serialisation which is incompatible with orthogonal persistence. A way around this is to require that the user defines methods for re-initialising native state when such an object is used. Unfortunately, this means that in this case the user must do more work in order to use persistence instead of less as is the ultimate goal.

PJama$_0$ did not support class evolution, but a tool for evolving classes offline [22] has since been developed.

2.4.4 JavaSPIN

The JavaSPIN project [7] aims to integrate Java with the existing SPIN architecture. The SPIN (Support for Persistence, Interoperability and Naming) framework [23] was developed as a foundation for integrating extended features with existing systems. SPIN has previously been used as a basis for integrating these features in extended versions of the C++ and CLOS APIs of the TI/Arpa Open Object-Oriented Database (Open OODB). The foundations for JavaSPIN are the SPIN framework, the Open OODB kernel and Java itself.

JavaSPIN's goal is to provide *seamless extension* of Java capabilities while ensuring *minimal barriers to adoption*. The extensions that JavaSPIN provides are persistence, enhanced name management and a basis for polylingual interoperability between C++, CLOS and Java. The persistence capabilities provided are reachability-based as well as name-based. Name management capabilities are independent of the persistence capabilities.

Specific ways used to make it as easy as possible for Java users to adopt the JavaSPIN extensions are the lack of language extensions, the lack of virtual machine modifications, unchanged core classes and no new native methods. Although store access currently requires native methods, but this will be changed so as not to require these.

There will be two related APIs for JavaSPIN. The first will provide basic persistence to Java users. The second will provide name management capabilities and name-based persistence for Java.

The appearance is that methods are added to the `Object` class and inherited from it by all other classes. The basic API supplies methods that can be called on objects to make them `persist` or `unpersist`. A name can be supplied when invoking the `persist` method which can be used to retrieve the object. If no name is given then the object can only be retrieved if it is referenced from another object. A `fetch` method is supplied for each class. This method returns the persistent instance of that class corresponding to the name that is given.

The second JavaSPIN API provides three more classes: `Name`, `BindingSpace` and `Context`. The `Name` class is provided for handling information about names for both transient and persistent Java objects. The `BindingSpace` class provides a directory abstraction for named Java objects. The `Context` class is used to form an appropriate context from one or more binding spaces and contexts.

In the JavaSPIN implementation strategy the additional methods that are defined are added to each class by modifying the Java compiler. Implementation of the additional methods are primarily done as native methods using calls to the Open OODB kernel and the name management and interoperability extensions.

In its original design, JavaSPIN did not allow class definitions to become persistent because there was no provision for class evolution. JavaSPIN has since been modified to allow classes to persist because class evolution support has beeb added [24]. This functionality is provided via the `replaceClass()` method which checks whether or not the new definition is a permitted modification of the old definition. This decision is based on whether or not classes that depend on the changed class will cause type errors and whether or not the new definition will cause compilation errors.

Some system classes and the JavaSPIN classes are always loaded from the file system and not from the store. This is intended to allow for the user to upgrade to a newer version of Java or JavaSPIN more easily.

2.4.5 Java Persistence by Checkpointing

An approach presented by Howell [25] provides persistence by checkpointing a running JVM process. By doing this all objects, class definitions and also thread stacks become persistent. The model does not allow for explicit transactions, but this was not required for the implementer's purposes which was to allow restarting of interrupted processes.

This approach relies heavily on the native platform's paging scheme to save and restore the JVM state. When this state is recovered, some external state can also be recovered, e.g. files can be reopened and communication links can be restored. Again, these are dependent on the chosen operating system.

2.4.6 Relational Database Access from Java

Work is also being done to allow data from relational databases and other legacy systems to be accessed by Java as normal Java objects. A few of these approaches are discussed here.

JDBC

JDBC™ (Java Database Connectivity) is a standard SQL database access interface, which provides uniform access to a wide range of relational databases. It is supplied as part of the standard Java distribution in the java.sql package.

JDBC is based on Microsoft's ODBC™ (Open DataBase Connectivity). It allows the user to do three main things: to connect with the database, to send SQL statements to the database and to process the results.

As can be seen in the example below, this creates many problems through the use of two distinct systems ("impedance mismatch" problems).

```
Connection con = DriverManager.getConnection (
  "jdbc:odbc:wombat", "login", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
```

```
while (rs.next()) {
  int x = rs.getInt("a");
  String s = rs.getString("b");
  float f = rs.getFloat("c");
}
```

Transparent Access to Legacy Data

The system proposed in [26] offers transparent access to data that is in the form of relations of tuples or files of records. The idea is to allow the user to view this data as collections of objects instead of in its original format. This is to be achieved via extensions to the existing PJama system.

External object faulters (XOFs) under the control of the PJama runtime system are used to wrap the legacy system. A prototype was built, but it deviated from the proposed design in some ways because the PJama source was not available. The extensions were thus all written in Java.

Accessing Relational Databases via Java Objects

The Persistent Java (PJ) system described in [27] uses a relational database as the storage layer for objects that are created within Java.

The main classes in the API are the `Database`, `Extent` and `Transaction` classes. The `Database` class provides for opening and closing databases and handling security and authorisation. The *extent* of a class is the set of all instances of that class that have been made persistent and the `Extent` class allows the user to scan this set. The `Transaction` class implements a simple ACID transaction model.

An object's `deepAccess` method can be used to load it as well as objects that it references. Its `deepUpdate` method will update it as well as objects that it references. Classes that are imported into the database must implement the `PersistentObject` interface. The import program generates implementations for the methods that are required by this interface, e.g. `access`, `update`, `delete`.

When a class is imported, a *class table* is created in the database in which to store fields for instances of that class. A table is also created in which to store static fields of that class. Tables are also created for storing arrays.

2.5 Other Persistent Languages

This section focusses on object-oriented languages as these are the most relevant to this thesis. However, the first language presented is Napier88, one of the best-known procedural persistent languages.

2.5.1 Napier88

Napier88 [15] was developed using experience gained from work done on PS-ALGOL [11]. It is a strongly typed persistent programming language supporting parametric polymorphism, first-class procedures, collections of bindings and abstract data types.

The Napier88 database is structured using *environments* to store collections of bindings between names and values (including procedures and other encironments). All data reachable from the top-level environment called `PS` will persist. Napier88 supports linguistic reflection (source code fragments can be converted to executable code at runtime) and functions can be stored in the database. The type `any` can be used to hold values of any data type.

A useful extension of the Napier88 Libraries consists of a collection of bulk types. One of these is a Map which is a set of *(domain,range)* tuples where *domain* values are unique. Creating a Map requires two procedures; one for ordering values and one for testing equality between values.

2.5.2 Persistent Modula-3

Modula-3 [28] is a strongly typed, object-oriented language that is based on Modula-2 [29] with a number of extensions. These include *thread support*, *exception handling*, *objects* and optional *garbage collection*.

In Modula-3, reference types may be either *traced* or *untraced*. Storage that is referenced by traced references is automatically reclaimed by the garbage collector when there are no more traced references to it. Any storage that is referenced by untraced references must be reclaimed explicitly when it is no longer required. Since the garbage collector needs to be able to find all traced references, untraced objects cannot contain traced references because they will be hidden.

To create persistent Modula-3, the type system was extended to include a *persistent* reference type. Any top-level variable may be declared as persistent and there must be at least one persistent variable from which other persistent data can be reached. In order for all persistent data to be found, untraced data may not contain persistent references. Conversely, persistent data items may contain non-persistent (traced or untraced) references. Such references will be set to `NIL` when the persistent item containing them is written to the store.

This design does not obey the principle of orthogonality, since to achieve orthogonality all data items should be allowed to contain persistent references and all items referenced from persistent items should be allowed to become persistent. On the other hand, keeping the distinction between persistent and non-persistent data is in keeping with the distinction between traced and untraced data that already exists in Modula-3.

Persistence is added to Modula-3 by extending the compiler. Object faulting is used, with residency checks inserted at appropriate points. A number of compile-time optimisations are used to eliminate unnecessary residency checks. These are covered in a later section of this chapter.

2.5.3 E

E [14] is a persistent, object-oriented language that is based on C++ [13]. There are three main extensions that have been added to C++ to create E. These are *persistent data*, *generator classes* and *iterator functions*. Only the first of these is relevant here.

Persistence is introduced into E by adding *db* types to C++'s type system. For example, to C++'s `int`, `char` and `float`, E adds the analogous db types `dbint`, `dbchar` and `dbfloat`. It also adds `dbclass`, `dbstruct` and `dbunion` which may only contain members which are also `db` types. A pointer to a `db` type object is also

```
dbclass person
{
public:
  dbchar* name;
  dbint birthyear;
  dbint age;
  dbchar* address;

  void changeaddress(char* newaddress);
  char* introduce();
};

/* method definitions... */

persistent person P;

main ()
{
  /* perform operations on P and the resulting changes will persist */
}
```

Figure 2: Declaration of a persistent object in E. The object's class is defined using the *db* types and the object itself is declared using the `persistent` storage class.

considered a `db` type, as is an array of `db` type objects.

To create a persistent object, the class must be defined in a `dbclass` definition with its members using appropriate `db` types. When a persisent instance of this class is required, it must be declared using the `persistent` storage class as shown in figure 2.

Dynamically created objects can be made persistent by allocating them inside a persistent *collection*. A collection is a simple persistence abstraction, viewed as a class that implements a persistent heap.

E's language design is not orthogonally persistent, but the programmer can achieve a higher level of orthogonality by programming exclusively with the db types.

E is implemented by extending the Cfront C++ front-end. Cfront converts C++

source to C source that can be compiled by a standard C compiler. In E 2.0, the EPVM (E Persistent Virtual Machine) is used. This handles access to the store and is invoked when necessary via a small set of C function calls that are inserted into the code produced by the E front-end.

2.5.4 C++ and O_2

The C++ interface to O_2 [30] allows the programmer to use all the features of the O_2 object-oriented database system from C++.

The C++ O_2 interface uses "twin" classes - a C++ class and an O_2 class to provide communication between the two systems. The C++ class "mirrors" the structure of the O_2 class in the database. The `import` utility is used to generate an O_2 class corresponding to an existing C++ class and the `export` utility is used to generate a C++ class corresponding to an existing O_2 class.

A *persistent pointer* is used to access persistent classes from C++. A persistent pointer is a special class that is generated when a class is either imported or exported. For example, a persistent pointer to an instance of a class called `X` would be an instance of the class `o2<X>`. To make programming easier, persistent pointers can be used in exactly the same way as standard C++ pointers in most situations.

O_2 databases have *named objects* as roots of persistence referred to from C++ programs. A temporary object will become persistent if it is reachable from an object that is already persistent. Objects can also be created directly inside a database by using a *handle* (obtained when a database is opened) with the `new` operator.

The system can run in a *read-only* mode or *transaction* mode. When a transaction has finished, the system returns to read-only mode so that it does not interfere with other transactions. A runtime library is linked into the C++ application when it has been compiled to allow it to communicate with O_2. It also makes sure that corresponding "twin" C++ and O_2 classes remain consistent.

2.5.5 Other Designs for Persistence in C++

A number of design considerations are examined in [31]. Two main issues are examined. The first of these issues is whether to integrate a high-level *DML interface* into C++ or to implement a lower-level interface in *class libraries*. The second issue is whether to add persistence to the language by using a persistent *storage class* or to use existing language constructs to supply an *abstract base class* of which all potentially persistent classes must be subclasses.

An example of the former is an embedded DML called OQL, designed to bring full OO-DBMS functionality and orthogonal persistence to the target language, with minimalist syntax.

Historically, systems using a DML embedded in a programming language have posed problems because of the discrepancy between the database types defined in the DML and the transient data types defined by the programming language. OQL overcomes this problem by allowing persistent objects to be created simply by using the `persistent` storage class with the standard programming language types. The DML preprocessor allows these persistent objects to be used in any expression or statement in the program.

Another approach described in [31] has persistence implemented "on top of" C++ by providing a virtual base class (in this case called `Pobj`) from which all potentially persistent classes must derive. This base class specifies methods that the programmer must implement as part of the class's functionality. Normal C++ pointers cannot be used to refer to persistent objects in this system; the `objRef` type must be used.

These factors mean that the programmer must work with two separate sets of data types - those for persistent data and those for transient data. Existing C++ code that is going to be used with persistent data must be modified to use `objRef`s where appropriate and class definitions must be extended to define the methods required by `Pobj`.

The C++/OQL integrated language proves to be both more efficient and easier to use than the base class approach, which moreover (by definition) only allows data of certain types to persist.

```
Let PersonObject = NewObject;

Let Person = IsA PersonObject With
  Name: String;
  BirthYear: Int;
  Age: Int;
  Address: String;
  changeAddress (newAddress: String): Null;
  Introduce: String;
End;

Let Student = IsA Person With
  Faculty: String;
  StudentNumber: Int;
  Introduce: String;
End;
```

Figure 3: Object and role declarations in Fibonacci. A new object type *PersonObject* is declared. A role type *Person* is then defined for *PersonObject*. A role subtype *Student* is declared, inheriting behaviour from the *Person* role. (Adapted from examples in [16])

2.5.6 Fibonacci

Fibonacci [16] is an *object-oriented*, *expression-based*, *persistent* language.

The system is interactive in that it constantly prompts for input at the *top level* of evaluation and returns results. Applications can also be run that interact with the top-level environment. Functions and types are both first class values in Fibonacci.

Fibonacci supports subtyping of objects with inheritance. Objects have a certain number of *roles*, which are similar to traditional object types. An object will behave differently depending on which one of its roles it is fulfilling when it is used.

Exception handling is also supported, as are *nested transactions*.

All data items that are accessible from the top level environment (and all items transitively reachable from these) will automatically become persistent. When an item is no longer reachable from the top-level environment, a garbage collector will dispose of it.

Fibonacci is an orthogonally persistent language; it allows all values to persist, even functions and types.

2.6 Introduction to Persistence Mechanisms

Once the language design has been finalised, mechanisms for supporting persistence are required so that persistence is transparent to the user. This section presents a brief introduction to persistence mechanisms before the later sections concentrate on specific strategies and optimisations.

An object must be resident in memory for it to be used; if it is not it must first be loaded from the persistent store. Commonly, two distinguishable forms of reference are used to differentiate between objects that are resident in memory and those that are not:

Local Address (LA) is a pointer into physical memory, used to refer to transient objects or to persistent objects that have been loaded into memory for use.

Persistent Identifier (PID) is used to refer to persistent objects in the store. The format of PIDs depends on the structure of the persistent store.

If an LA is encountered then execution should not be interrupted. If a PID is encountered then steps have to be taken to load the required object (called an *object fault*, similar to a page fault in operating system parlance). Firstly, we perform a *residency check* to determine whether or not the referenced object is already resident in memory. If it is already resident then we do not have to load it again, so we call this a *false object fault*. A depiction of the incremental faulting process is given in figure 4.

When an object is loaded, the mapping between its PID and its LA is added to a table to facilitate future residency checks. To benefit future uses of a reference, its PID can be overwritten with its corresponding LA to avoid unneccesary residency checks. This practice is known as *swizzling* and comes in a variety of flavours, many of which we encounter later in this chapter.

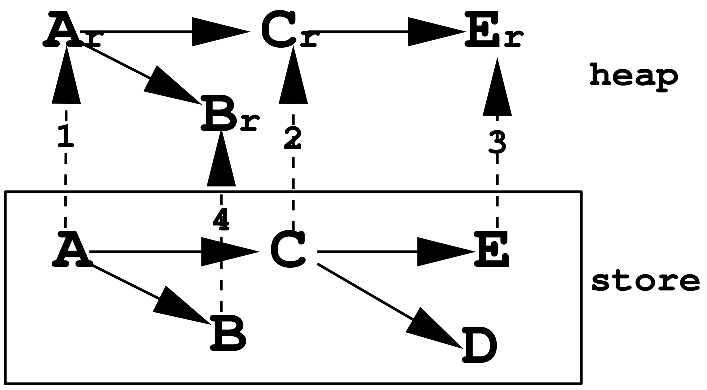

Figure 4: Incremental loading from the store. `A` is the first object that is used and it is made resident in memory as `Ar`. A reference to object `C` is then used and it is made resident as `Cr`. The same then happens for `E` and `B` in that order. No reference to `D` is ever used, so it is never made resident.

A PID may use the same format as an LA if object faults can be detected by the address translation unit's protection mechanism. Systems that use this method will not be portable due to their dependence on hardware support.

Storage of data into the persistent store also has to be automated. When an application requests a checkpoint, all *dirty* resident persistent objects must be written back to the store. Also, any new objects that have been created and have become reachable from any persistent object need to be *promoted* to persistence. This process is depicted in figure 5

2.7 Optimisation Techniques

2.7.1 Swizzling Strategies

The first extensive study on the effects of different swizzling strategies is presented in [32]. This study compares different swizzling schemes in the context of the Mneme system. A load-work-save cycle is used as the application model since it is typical of an edit session in e.g. CAD applications.

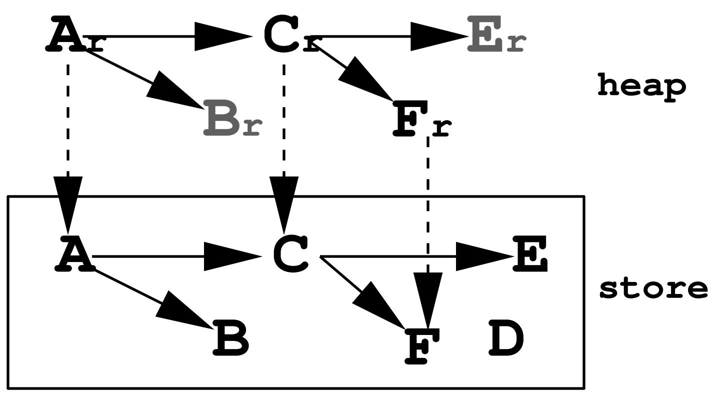

Figure 5: Writing back to the store. The grey objects have not been updated and therefore do not have to be written back. `Cr`'s reference to `Dr` has been overwritten with a reference to the new object `Fr`. `Fr` is put into the store as `F` and `D` is no longer reachable from anything else in the store, so it will be disposed of by the garbage collector at some stage.

There are three basic approaches. The first of these is *don't swizzle*. The second swizzles *in-place* which means that the objects are left in the object cache and modified there. The third approach is to make a *copy* from the object in the cache when swizzling it. In-place swizzling saves on the cost of copying objects, but these objects all have to be deswizzled when they are written back to the store instead of just those that have been modified.

With in-place or copy swizzling, either *lazy* or *eager* swizzling may be done. Eager swizzling swizzles all required objects in advance whereas lazy swizzling only swizzles objects when they are needed.

Experiments showed that lazy swizzling always costs more than eager swizzling, up to 16% to 18% more in loading. Copy swizzling costs more per byte but less per object than in-place swizzling. This higher cost per object (in-place) comes from the additional 4 bytes that are needed per object to support deswizzling and if the cost of loading and writing these extra bytes is removed then in-place swizzling is a little faster than copy swizzling.

There are space overheads when swizzling, and an advantage when references are followed. Executions using lazy swizzling can take as little as 74% of the time that non-swizzling does and eager swizzling as little as 68%.

[33] investigates a portable and efficient method for accessing persistent objects in the context of the E programming language [14, 34]. This paper introduces an alternative E Persistent Virtual Machine (EPVM 2.0) specifically for CAD applications where the load-work-save cycle is common. The work phase usually involves editing which requires repeated traversals of the data.

EPVM 2.0 maintains a cache of objects that the E program has used. The cache uses a scheme that swizzles inter-object references when their location is "discovered". Because a pointer that is discovered might not be dereferenced, the object that it identifies may not need to be loaded. To avoid unnecessary loading of objects, a "discovered" pointer is swizzled if the object is already cached. Otherwise, objects are loaded into the cache only when their PIDs are dereferenced.

A number of experiments aimed at evaluating the effectiveness of EPVM 2.0 were conducted using the OO1 benchmark suite [35]. A client-server model was used for the experiments and comparisons were drawn with three alternative systems. These were ObjectStore V1.2 (that uses a memory-mapped approach to faulting and swizzling), EPVM 1.0 (which supports a limited form of swizzling) and direct calls to the Exodus Storage Manager (ESM) by a non-persistent language such as C++. The results compare the efficiency of the different schemes for faulting and swizzling (including no swizzling). Two different caching schemes were also tested with EPVM 2.0: object caching and page caching.

A large number of results are given, including usage for small and large databases and different caching techniques and cache sizes. The results are summarised here: In the general case, EPVM 2.0 outperformed both direct calling of the ESM and EPVM 1.0. In the case of small databases, EPVM 1.0 outperformed direct ESM calls but the opposite was the case with large databases.

2.7.2 Reducing Residency Checks

Another study by Moss and Hosking presented in [36] deals with a number of optimisations for Persistent Modula-3 and Persistent Smalltalk which require complex changes to the compiler. Ideas involving residency checking include:

Local or global subsumption If there is a number of uses of an object in straight-line code then the first will enforce residency, assuming that it is pinned. (Pinning requires objects that have been faulted in to remain resident for as long as they are referenced by registers or local variables.) Local subsumption is concerned with enforcing this within basic blocks, while global subsumption can be used between basic blocks by performing data-flow analysis.

Target object residency This involves making the *target object* in object-oriented method invocations resident, thus doing away with the need for residency checks on it during the execution of the method.

Formal parameter residency It can help the execution of a method or procedure to know if its parameters are resident prior to invocation. In this case, no residency checks are needed when the parameters are used.

Procedure result residency If a procedure returns an object that is resident it may help if subsequent code knows about it.

Data type assertion If a multi-level pointer is used it may be helpful to fault in several levels of objects at once. This might cause many objects to be loaded in the case of complex structures, but if those objects are going to be used then it might be a good idea.

Hosking examines methods of reducing the number of checks that are required for faulting in [37]. Possible execution scenarios are given and performance tests are run with a persistent Smalltalk implementation. Three execution scenarios were considered. These are:

Procedural All procedure bindings are statically determined. Every reference requires a residency check unless data flow analysis has been performed.

Static OO When one of an object's methods is executed, that method typically makes use of the object that encapsulates it. If the target residency rule is applied then all residency checks for the target object can be eliminated. This means that object residency must be checked before execution of the method is started. If the target object is the same as the calling object then it will already be resident.

Dynamic OO Because object oriented languages support inheritance and polymorphism, the actual method that is invoked at a certain point in the program may change during the execution depending on the type of the target object. The residency check for the target object can be built into the dynamic method invocation mechanism to ensure target residency.

The OO1 benchmarks [35] were used to compare the execution scenarios. The database consists of 20000 part objects, each of which has connections to three other parts. The benchmarks consist of three separate operations that use the database: lookup, traversal, insert. An update [33] test is used to give a better measure of update overhead than the insert test. Checks due to checking for residency of processes, stacks and other objects that can persist in Smalltalk are ignored to obtain a close analogy to more traditional languages such as C, C++ and Modula-3.

In the procedural scenario, no residency checks could be avoided. In the OO scenarios, the target residency rule allows a significant number of residency checks to be avoided: The static OO scenario requires residency checks on target objects that are not the same as the current target object and can eliminate 24%-75% of checks, depending on the benchmark test. In the dynamic OO scenario, residency checks for target objects can be performed as part of the method invocation mechanism and could eliminate 86%-99% of checks.

2.7.3 Caching

Caching in PJama

In order for execution to continue within limited physical memory, memory has to be re-used. In PJama the page buffer pool (PBP), object cache (OC) and garbage-collection heap (GC heap) all perform memory recycling. An object can be recycled if it is no longer reachable from any executing threads or if it is not likely to be used again.

The OC needs to provide space for all allocation requests. For efficient operation, it needs to retain as many currently resident objects as possible and needs to optimise the locality of objects. There are problems using a standard garbage collection algorithm for managing space recycling in the OC, since objects that are still reachable might need to be evicted in order to free more space.

The OC is divided into a static area and a dynamic area. The static area is used for certain housekeeping functions and for the bootstrap region which stores certain core classes. The dynamic area is divided into regions, each with objects of a certain kind and a *region manager* that is adapted to managing objects of that kind.

The unit of OC space reclamation is the region. When available OC space reaches the "low-water mark", asynchronous reclamation is started and is stopped when the free space reaches the "high-water mark".

Dangling pointers will be created if an object that has direct pointers to it is evicted from the OC. All locations on the stack that might contain direct pointers to OC-resident objects are recorded. When the OC needs to pin objects that are accessed by threads it scans the handle-variable stack of each thread and pins each referenced object that is in the OC.

Two methods of implementing LRU victim selection are suggested. The first requires "hiding" objects by overwriting their object pointer with their PID while storing the object pointer in the ROThandle. When a hidden object is accessed, an object fault will occur which is handled by copying the PID and object pointer back to their original positions. When victims are required, the set of objects that remains hidden is the LRU set.

The other method flags objects accessed during a set time period. After a number of such time periods, the objects that were flagged during the least recent period is the LRU set.

The presence of pointers or handles on the native or Java stacks could mean that the referend objects are likely to be used again. Threads that are scheduled for execution soon can also be used to determine which objects are likely to be used soon.

Buffer Management and Persistence Optimisations

Cutts, Lennon and Hosking [38] seek to introduce cooperation between the buffer manager and the execution engine, two modules that are traditionally kept separate for the most part. Running multiple threads, each with its own object faulting requirements but using a single buffer, means that static analysis alone is not enough to manage the buffer efficiently.

The pattern of object access is analysed statically so that subsequent accesses of the same object can assume that it is resident. The optimiser rewrites the code and this is only safe if certain guarantees on the state of the objects in the buffer can be upheld.

The first of these guarantees is *guaranteed object residency* which requires that objects in use be pinned in the buffer. The second is *guaranteed object marking* which requires that objects that are the subject of updates that have been optimised out retain their updated status.

If these guarantees require large numbers of objects to be pinned then the buffer manager's ability to free up space for new objects can be seriously impacted. If the set of pinned objects and the objects required to maintain a good working set overlap significantly, then the adverse effects on buffer space recycling can be minimised. Since the interaction of multiple threads is unknown at optimisation time, the optimiser can specify a minimum pinning level to the run-time system. The system is free to do further pinning above this specified minimum.

Objects reachable from the "hot" area at the top of a thread's stack is likely to represent a significant portion of that stack's working set. A guarantee to retain these objects across buffer recycling operations can be made. In this way, the pinning area moves as the stack grows and shrinks.

2.8 Faulting Strategies

2.8.1 Object faulting in CPOMS

The CPOMS Persistent Object Management System [39] was written for use with the PS-ALGOL system and is a good early example of a system that uses object faulting.

In the PS-ALGOL interpreter, PIDs are the same size as local addresses (LAs). This means that a PID can easily be overwritten with an LA. Each object has a unique PID and all PIDs have their high bit set so that they can be identified. This check is done by software and when a PID is encountered, the object that it references needs to be loaded from the store.

To prevent multiple copies of an object being created, a table called the PIDLAM (PID to LA Map) is maintained. Since PIDs are swizzled, the LAs have to be converted back to PIDs when they are written to the persistent store. This means that the PIDLAM must support queries by both PID and LA so that when references are deswizzled for writing to the store, an LA's corresponding PID can be found.

The PIDLAM has both permanent and temporary entries. New objects are added as temporary entries until either a commit or an abort is requested. When a transaction starts there should be no temporary entries in the PIDLAM. During the transaction, all the new objects that are created will be temporary. If an abort occurs, none of those objects should be recorded and all the temporary entries in the PIDLAM are discarded. If a commit occurs then the temporary entries need to be changed to permanent entries, including all those objects that are reachable from the objects to which they correspond.

2.8.2 Object Faulting in PJama

PJama, an orthogonally persistent implementation of Java [40], was developed at the University of Glasgow in collaboration with Sun Microsystems, extending Sun's Java Virtual Machine (JVM).

The architecture of the persistent machine was designed to keep the persistent objects that have been faulted into memory looking as similar as possible to the conventionally created objects used by the unmodified JVM. This approach would mean that fewer changes would be required in other areas of the JVM to support persistence. With this in mind, the following decisions were made:

- The memory system uses two levels of buffering. These are a Page Buffer Pool (PBP) which holds pages taken from the Persistent Object Store (POS) and an Object Cache (OC) which stores faulted objects that have been taken from the PBP and converted to the JVM's internal format.
- All threads running in the JVM share the memory areas (PBP, OC and garbage collected heap (GC heap)).

In the JVM architecture, Class instances and arrays are allocated into the GC heap. Other data such as class objects and stacks are allocated outside the GC heap. Objects and class objects are each given a *handle* by which they are identified for the duration of their lifetime in the JVM. A handle contains the address of the relevant object and a pointer to the object's method table and class descriptor. Handles are stored separately from objects in the heap.

Because all objects in the JVM are 8-byte aligned, the lowest three bits of their addresses will always be zero. PIDs can be identified because they will have one of these bits set. If one of these bits is set then the required object must be faulted in. In order to make the faulting thread-safe, a mutual exclusion lock is acquired for the handle.

Any persistent objects that are needed by the JVM are loaded on-demand by the OC. The OC retrieves the object from the store layer and swizzles the references that it contains, if any. *Indirect swizzling* is used because of the handle system.

Eager indirect swizzling is used for class instances: since these are usually small, swizzling all their pointers is not expected to be too expensive. Arrays are potentially very large and all their elements might not be accessed, so lazy swizzling is used.

A *resident object table* (ROT) is used to store mappings from PIDs to handles and is hashed on PIDs to make searches on PIDs efficient. Handles stored in the ROT are called ROThandles.

A *persistent class directory* (PCD) is maintained by the OC, containing mappings from class names of all the classes in the POS to the class descriptors. Special mechanisms are needed for faulting in classes and their associated methods' bytecodes. If a requested class is not found in the PCD then it is loaded from the file system as for conventional Java classes. New objects created by the JVM via bytecode instructions are stored on the GC heap as usual.

Four types of references are used. There are two main types of handles, *Jhandles* and *ROThandles*. Standard Jhandles are in the format that the unmodified JVM uses; the handle contains a pointer to the object and a pointer to its method table. Jhandles for instances of persistent classes are used when the JVM creates a new instance of a class that is in the POS; the handle and object are allocated as usual, but the method table to which the handle points is in the OC.

A *fault block* is used during indirect swizzling. It is a ROThandle whose object pointer is a PID. Dereferencing this pointer will cause an object fault and the required object will be loaded. *Swizzled ROThandles* are superficially the same as standard Jhandles but are postfixed with extra information that the ROT uses. The object pointer points to the object in the OC and the method table is also resident in the OC.

Each object in the OC is prefixed with an index into the ROT so that its handle can be found. The lower three bits are not used as part of the index but are used to store information about updates.

The first step when objects are being faulted in is to find the size of the object. The store layer has an interface to find the size of a particular object. Then a copy of the object is obtained from the store layer and finally the object is prepared for use by the JVM.

2.8.3 Page faulting

The above process of loading individual objects, either when or before they are needed, is known as *object faulting*. A distinction is made between this and *page faulting* where every object on a database page is loaded at the same time. While object faulting is usually simpler and easier to visualise, page faulting can have performance advantages if the hardware or operating system can be used to facilitate it.

A system for pointer swizzling at page fault time is presented in [41]. This helps to preserve object identity in systems that attempt to provide huge address spaces for distributed shared memories, shared address spaces and persistent object stores. A problem arises because the large number of objects might exhaust the available hardware-specific address bits. Pointer swizzling can be used to avoid this problem by converting long-format pointers into physical memory addresses when they are needed.

In conventional swizzling schemes a persistent pointer is swizzled when it is used, which can require a large number of software checks. The scheme presented in [41] seeks to avoid this overhead while still allowing the system to run on conventional hardware. All schemes that use incremental faulting of objects need to use some sort of check to detect references to persistent objects so that they can be faulted in before they are used. The authors of [41] chose to use standard virtual memory hardware's access protection capability to achieve this.

If the program tries to access a page that might contain persistent pointers then a trap handler is invoked and all of the persistent pointers in that page are swizzled. Instead of just faulting in objects, the whole page that contains the required object is faulted in. This reduces the size of the tables used to record mappings between persistent and transient addresses and causes the effect of a "wave front" of address translation that swizzles all persistent pointers before they are encountered by the program.

Since virtual memory is reserved for a page when a persistent pointer into it is found, the virtual address space could be exhausted if there is a large number of pages that contain almost exclusively pointers.

A fraction of each virtual page can be used, but they can be arranged so that more than one virtual page shares the same physical page. While this means that each physical page has more than one virtual page number, the nonoverlapping allocation pattern will ensure that no objects are aliased. This scheme will cause fewer pointers to be discovered at one time (and therefore fewer virtual memory pages to be reserved), but it actually increases the total amount of the address space that is used because it decreases the amount of each virtual page that can be used.

A hybrid scheme suggested in [42] seeks to avoid software checks during a dereference (as in CPOMS) and also to avoid the greedy allocation of virtual memory pages (as in Wilson's scheme [43, 41] above).

This system requires that PIDs are at least twice as large as LAs. For the purpose of discussion, LAs are assumed to be 32 bits and PIDs 64 bits. Page faulting is used as in Wilson's scheme, but instead of eagerly reserving memory for pages that are referenced, a two step swizzling method is used. A reference points to either:

- an entry in a translation table if the referend object is not present in memory (partially swizzled)
- an LA if the referend is resident (fully swizzled)

A translation table in this scheme may use either a *per object* table (e.g. CPOMS) or in a *per page* table (e.g. [41]). When a partially swizzled pointer is dereferenced an access fault occurs and triggers the second phase of the swizzle.

In the second phase the partially swizzled pointer is overwritten with a virtual memory pointer to the object in memory. If that object is not in memory then the page containing it must be loaded from the store. To do this, the PID from the translation table is used. The page load may result in new entries being made in the table. Unlike in Wilson's scheme, it is only when an object is used that memory is allocated for it. When a PID is overwritten with an LA, only half the space in the PID is actually used. The other half can be used to point to its entry in the translation table to facilitate deswizzling.

In Wilson's scheme all pointers on a loaded page point to correct virtual memory addresses because *eager swizzling* is done. This can be simulated to a certain degree in the hybrid scheme. A linked list called the *partially swizzled pointer chain* can be used to join together all instances of pointers referencing objects on a page. When an object is faulted into memory the swizzling code not only swizzles the pointer that caused the fault but follows the pointer chain and swizzles as many other pointers as it can. This addition to the scheme requires that PIDs be large enough to hold a "next" pointer for the linked list and that a per object instead of a per page translation table be used.

2.8.4 Faulting in Other Architectures

Testing different caching schemes for EPVM 2.0 [33] revealed that page caching gave better performance than object caching. When "warm" (a number of objects already cached) and "hot" (the asymptotic case - as many objects as possible already cached) traversals were performed, ObjectStore outperformed the other systems because its memory mapped architecture meant that it needed no extra to check whether or not pointers were swizzled. In the "cold" case (no objects cached), EPVM 1.0 and direct ESM calls outperformed EPVM 2.0 and ObjectStore.

The Mneme architecture described by Moss and Hosking in [44] is similar to that described in [33]. Objects are copied from the storage manager's buffer pool and in the process they are converted to the form required by the program (including swizzling). Objects are transferred from disk in groups, which improves on the LOOM [12] system which used individual object faulting.

The heap is viewed as a directed graph, the objects being nodes joined by references which are the edges. Two different methods of detecting the need for object faults are *edge marking* and *node marking*.

Edge marking involves marking individual references.

Node marking involves converting all references in an object to memory addresses when the object is loaded. A small proxy object (or *fault block*) can take the place of an unloaded object. These proxy objects can be distinguished from normal objects and when one is accessed, the object that it represents is loaded and the proxy object now points to the loaded object and is called an *indirect block*.

The OO1 benchmarks [35] were used to compare different faulting schemes. Schemes that used fault blocks and faulted in objects one at a time were markedly less efficient than those that loaded objects a page at a time. Fault blocks that were allocated in protected pages were very slightly less efficient than manually checked fault blocks. In schemes that used edge marking, all the options were slower than the node-marked schemes and those that loaded many objects at once were more efficient than loading one at at time.

1	partition number	object number
1 bit	15 bits	16 bits

Figure 6: Structure of a CPOMS PID

2.9 Persistent Store Architectures

2.9.1 Introduction

Three store architectures are covered here. The first two are the Persistent Object Management System written in C (CPOMS) [39] and the Persistent Java Store Layer (PJSL) [8]. The CPOMS supported the first persistent language and its design formed the basis of the PJSL store. PJSL is covered because it has been used with the PLaVa machine. PSI [5] is then sketched as an example of an alternative approach.

2.9.2 The CPOMS Store

The CPOMS (Persistent Object Management System written in C) store [39] was implemented for the PS-ALGOL interpreter [11]. An index file contains mappings of object numbers to locations in the database file. This allows object-level addressing and also simplifies the process of compaction when the store is garbage collected. Multiple databases of varying size are supported.

The store comprises a number of partitions: each existing partition has an entry in a global partitions file, mapping it to a certain database. The first partition allocated to a database has as its first object the root object of the database. Locks held on the database are recorded; CPOMS uses a one writer or many readers locking protocol.

The internal support needed for this scheme includes two data structures. The first is a list containing entries for each database that is currently open. The second is used for identifying a database and an index file to use for each partition. These two structures need to be duplicated for each PS-algol program that is run.

2.9.3 The PJama$_0$ Store

This persistent store was the precursor of PJSL [8] and was built using the *Recoverable Virtual Memory* [45] (RVM) system from CMU to implement the persistent store. RVM imposes a no-steal policy, so no pages containing uncommitted updates could be flushed to persistent storage.

The store layer provides an interface for performing the following tasks atomically: loading persistent objects, writing back updates on a set of objects, storing a set of new objects and creating, opening and closing stores. This functionality is encapsulated, along with the PBP. When pages need to be removed from the buffer, a two-hand clock algorithm is used to select victims. Only pages that have no uncommitted updates can be evacuated due to RVM's no-steal policy.

2.9.4 The PJSL Store

PJSL [8] is a persistent object store based on experience with the prototype PJama$_0$. It is designed to permit incremental store management and to allow a number of object management regimes within the same store. The design of a new store was brought about by the fact that the old store was too heavily dependent on the implementation of a particular version of the virtual machine.

The design goals are the following [8]:

- to support all object types
- to accommodate at least 10 GB of highly structured data, typically dominated by large numbers of small objects
- to be capable of continuous operation
- to be capable of running with a minimum of operating-system dependent code
- to be appropriate for long-running transactions, archiving and distribution.

PJSL adopts a partitioning scheme which allows each partition to be garbage collected separately. Management of free space within a partition can be achieved in

Figure 7: Structure of a PJSL PID

a number of ways. The management scheme that is chosen, together with a number of other organisational parameters is be referred to as a partition regime. Different partition regimes are appropriate for objects of different size, structure and behaviour; and partitions belonging to different regimes can co-exist in the same store.

Objects in PJama can be divided into four main *kinds*: instances, arrays, byte-code arrays and class objects. The object cache manager specifies the regime under which an object has to be stored and the store layer handles the allocation, reference counting, garbage collection, etc.

The PID is made up of a 32-bit word comprising partition ID and object number. This is calculated to provide enough address space to support the planned maximum 10GB store.

A partition ID stays attached to that partition until it becomes empty and is reclaimed. This way, the partition can be moved and the PIDs of its objects will remain valid. Each partition contains an *indirectory* which maps object numbers to offsets in the object space section of the partition. Because of this, if an object is moved inside a partition due to compaction its PID stays the same. Both intra-partition and cross-partition references go via the indirectory.

A *descriptor* is used to record where the references in an object are so that they can easily be found. Not all objects need descriptors, e.g. scalar arrays don't need them because they contain no references. All instances of the same class use the same descriptor and the descriptor contains a reference to the class object.

Two levels of free space management are needed: one to allocate partitions inside the store and another to allocate objects inside a partition. The store is split up into small blocks (SB) of a fixed size and partitions are made up of numbers of these.

Once a partition has been allocated, the space inside it is managed at the object level. Because of the use of partition regimes, different partitions can use different management policies, optimised for the particular kinds of objects that they contain.

2.9.5 Persistent Store Interface

A Persistent Store Interface (PSI) [5] that uses a transactional cache is presented in [46] as a basis for implementing a persistent Java implementation.

The interface is split into *core* and *extended* functionality. The core interface provides functionality for basic ACID transaction caching. The extended interface provides features such as advanced storage structures, intra-transactional checkpointing and rollback.

Three main ideas are looked at separately in order to create a clear abstraction of the transactional object cache architecture. These ideas are *stability*, *visibility* and *cache management*. The first two are essential elements of a transactional storage interface and the third is added to provide efficiency. These three ideas were treated as orthogonal when models for them were developed and were then combined to form the transactional object cache architecture.

The PSI core interface is defined in terms of the semantic framework. The calls are: `PSI_Read` to read an object into the cache, `PSI_Write` to update an existing object, `PSI_New` to create a new object, `PSI_NewTransaction` to start a transaction, `PSI_Commit` to commit a transaction, `PSI_Abort` to abort a transaction. `PSI_Unfix` makes an object unavailable and `PSI_Fix` makes an object available if `PSI_Unfix` has been called on it since a `PSI_Read` or `PSI_Write` call.

Six key issues in the design of persistent programming languages (PPLs) are identified. The extent to which PSI will play a part in these aspects on PPL design is determined. The aspects are:

Persistence Identification A PSI store supports persistence by reachability from a single persistent root. The client PPL may use other *named roots* in the database, but it is the client's responsibility to make sure that these roots remain reachable from the store's root. For the sake of efficiency, PSI uses *descriptors* to record the location of references in objects.

Residency Checks and Write Detection Because residency checks and update tracking can be done in so many ways, the PPL implementation has to make explicit read and write requests to PSI.

Swizzling PSI supplies hooks that facilitate swizzling but does not enforce any particular swizzling strategy because the choice is strategy depends greatly on the client's needs.

Concurrency Control and Cache Management PSI supplies the client with tools to implement a wide variety of transaction models while leaving the store designer with freedom of implementation.

Recovery The notion of durability for ACID transactions is built into the PSI core. PSI's logging extension supplies intra-transaction stabilisation and rollback facilities.

Advanced Storage Structures Because some clients might require extended storage structures, index and collection types have been implemented in one of PSI's extensions to provide concurrency control and storage management optimisations.

2.10 Conclusion

2.10.1 Using Persistent Java

Experiences using PJama_0 with example applications are presented in [20]. The first sample application tested was a GIS system called Oscar. The information is displayed using the AWT. Converting the Java version to PJama involved changing 23 lines of code. Loading the data from files in the Java version would take between 3 and 8 minutes per tile. The PJama version took 20 seconds to load, process and display a tile.

Forest is a CASE application, the first prototype of which was written in C++ using the ObjectStore database. The PJama version differed from the C++/ObjectStore version in some areas of implementation, it was not a direct port since the C++/ObjectStore version used a client/server model. When the benchmark was run as a single transaction it ran in 53 seconds with C++/ObjectStore, while the PJama version ran in 12 seconds.

The Java compiler was also used as a sample application. Classes that are loaded from the persistent store do not need to have their bytecodes re-optimised. Another advantage is that the compiler error messages do not have to be loaded from the properties file when the compiler starts. Running times for compiling all the classes in the `java.lang` package are given here:

java	pjava	pjava-s	pjava-sb
380	392	332	346

The "java" column represents the time taken by the standard JDK 1.0.2 interpreter, "pjava" is for PJama running in non-store mode, "pjava-s" is for PJama running in store mode with no bootstrap classes and "pjava-sb" is for PJama running with all classes in the bootstrap region.

Jeeves is an HTTP server written entirely in Java. The Jeeves server ran 6% slower under the PJama virtual machine in non-store mode. Since multiple threads might attempt to stabilise the store at the same time, stabilise calls had to be serialised in PJama_0.

Overall, the early experiences with PJama were positive both in performance and reliability. Even with the simple global stabilisation that was available, many useful applications could be run.

2.10.2 Current and Future Work

Much current work in the area of persistence is being done with Java because it is widely supported and is well suited to persistence. Some of this work is devoted to adding features to persistent Java to make it more useful. A particular area of interest is that of supplying persistent Java with a useful transaction model [47, 48, 49].

Using Java to access legacy data such as data stored in relational databases is another topical issue and is addressed in [26, 27]. This idea has been expanded in [50] to include both relational databases and object-oriented databases.

The issue of persistent store management suitable for persistent Java is being studied [8, 51] as well as suitable cache architectures [40, 46]. The problem of class evolution for stored classes is tackled in [24] and [22]. The issue of a distributed version of persistent java is discussed in [52].

Chapter 3

Non-Persistent JVM (LEAN)

3.1 Introduction

This chapter describes the LEAN [53] machine, the lightweight JVM on which PLava was based. Although this JVM existed in a useable form at the start of this thesis, so many extensions and improvements were made to it that very little of the original machine remains. Besides persistence mechanisms, other changes include better memory management and garbage collection, better thread support, JNI support [54] and an improved class loader. For the sake of clarity, this chapter describes the machine before persistence was added to it, rather than the original LEAN which it replaced.

I developed the LEAN (Lightweight, Embedded, Architecture-Neutral) Machine as an Honours project to investigate the feasibility of implementing a JVM on a small machine such as a digital satellite television (DSTV) decoder, with 1 megabyte each of ROM and RAM. This had also to be as portable as possible. Thus LEAN was designed to give adequate performance on a small machine without using any hardware- or operating system-dependent primitives.

The motivation for doing this arose from the appropriateness of Java for embedded applications because of its platform independence and its safety and security features.

Initially, the modules of the virtual machine were implemented in a basic system based on the Java 1.0 specification to demonstrate the feasibility of the project. The system was then improved to gain better performance. A few methods of optimisation were implemented and evaluated. The system was made suitable for installation in ROM on the DSTV decoder. This included pre-loading and pre-optimising Java classes that needed to be used.

The chapter addresses the portability issues first, then gives an overview of the Java virtual machine before the main section which deals with the implementation of the virtual machine. This is followed by an account of the optimisations that were incorporated.

3.2 Portability Issues

To ensure portability of source code, only ANSI C constructs and function calls were used. No operating-system specific calls were made in the virtual machine. Specifically, threads were implemented entirely internally to the virtual machine. Some operating system calls are unavoidable in the implementation of native methods (e.g. file handling) and these will need to be changed if the system is ported to a platform that doesn't support these calls.

In order to ensure portability of data types, a set of data types was made that would be consistent across all platforms. Problems arise due to the fact that C integers are 16 bits large when compiled on a 16-bit compiler and 32 bits when compiled on a 32-bit compiler. Situations like this have to be avoided because the sizes of data types must be consistent across platforms once they have been defined. In order to achieve this, a set of standard types was defined in a header file which is included in all modules. This header file includes pre-processor directives to ensure that the types are of the correct size whenever they are compiled.

3.3 Overview of the Java Virtual Machine

The Java Virtual Machine [55] (JVM) is the "machine" on which the compiled Java bytecode is executed. It must provide an interpreter for the Java bytecode and support for all the Java language constructs such as threads and exceptions [1].

The JVM is a stack-based machine. Space for the operand stack and local variables is allocated on the Java stack which also contains current execution information, similarly to a call stack frame in many languages.

For increased efficiency, Java bytecode can be compiled to native machine code for the platform on which the virtual machine is running. This is often done by compiling the bytecode before the method is executed, which is known as just in time (JIT) compiling. Virtual machines making use of JIT compilation have shown notable speed increases [56], but this is obviously platform dependent and was not considered in this project since portability was a priority.

Because the Java bytecode is portable and interpreted by the virtual machine, some other mechanism has to be provided to give applications access to platform-specific services. These services include file I/O, any textual or graphical output and network access. To provide access to these services, Java allows native methods to be run. These are methods written in C that are called from the JVM. The Java programmer calls them in the same way as Java methods and the JVM must check method calls to see if the method being called is a Java or a native method and do the invocation accordingly.

3.4 Virtual Machine Implementation

3.4.1 Bytecode Interpretation

The bytecode instructions can be broken up into a number of categories: manipulating the operand stack, local variable access, array and object creation, field access and method invocation. Most of the instructions take a fixed number of operands but some (e.g. method invocation) take a variable number of operands. Instructions are executed in the interpreter loop and may make use of the registers, operand stack and local variables. Each instruction's required behaviour is specified in the Virtual Machine Specification [55].

3.4.2 The Java Stack

A stack frame in the Java environment stack includes all the data needed for executing a certain method. It stores the values of the virtual machine registers as well as other data such as the current method's local variables and the operand stack. A separate Java stack is maintained for each executing thread. A pointer to the "previous frame" is stored in each frame and is null in the first frame of every thread.

The virtual machine specification states that stacks may either be of fixed size or of variable size. LEAN has variable size stacks using the following scheme: stack frames are allocated from a large chunk of memory called a superframe. If the current superframe is full when another frame is required, another superframe is allocated. When the first frame in a superframe is popped, the superframe is freed. Other than these checks, the traversal of the stack between frames within a superframe and frames whose links go between superframes happen transparently because the "previous frame" pointer identifies previous frame each time, whether or not it's in the same superframe.

Initially, memory was allocated each time a frame was required and de-allocated when it was popped for the sake of simplicity. Marked performance improvements resulted from the superframe scheme.

Operand Stack

The operand stack is a number of 32-bit "slots". The size that a method's operand stack needs to be is specified in its class definition and that amount of space is reserved in the method's Java stack frame. When a long or a double value is pushed onto the operand stack it takes up two consecutive slots because these two data types are 64 bits in size.

Local Variables

Like the operand stack, a method's local variables are stored in its stack frame. Each local variable slot is 32 bits wide and 64-bit data values are stored in consecutive slots.

3.4.3 Registers

The JVM has four "virtual registers": the program counter, the pointer to the top of the operand stack, the pointer to the local variables and the pointer to the current stack frame. These values are mirrored in methods' stack frames so that they can be saved and restored as required.

3.4.4 Heap Management

The Java heap is shared among all threads, unlike stacks which each belong to a particular thread. All class instances and arrays are allocated from the heap. If there is not enough heap left to satisfy an allocation request then the heap is garbage collected. If there is still not enough free space then the heap is expanded. The heap will not necessarily be contiguous in memory (this is not a requirement in the virtual machine specification).

Since the Java language does not provide any means for explicit de-allocation of memory, some sort of automated heap management is required if applications are to run in a finite memory space. We use a garbage collector to discard any objects or arrays that are no longer reachable by the application, since they cannot be used.

Allocation

In LEAN, the heap is organised as a linked list of heap blocks and memory allocation requests are satisfied by reserving unused space within one of these heap blocks. If a bigger heap is needed then the heap is expanded by adding another heap block onto the end of the list. A "first fit" scheme is used for allocating the requested memory. After the garbage collector has been used, there will be free spaces between the allocated spaces. These are linked together to form a "free list". Each heap block has an associated bit map containing one bit for each 32-bit word in that heap block. When memory is allocated in the heap block, the bit corresponding to the beginning of that section of memory is set. This bit map can be used to find the starting addresses of objects in memory.

Since Java objects and arrays have a certain minimum memory usage (i.e. the

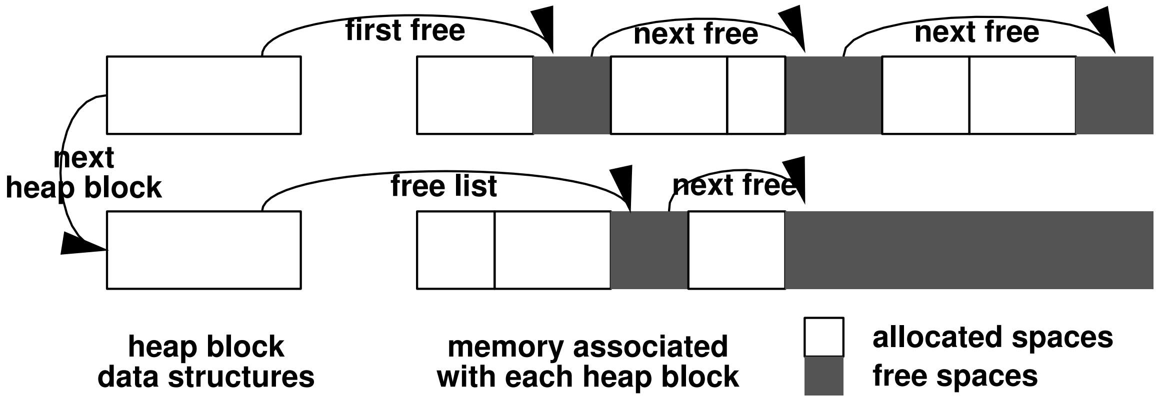

Figure 8: Hypothetical memory usage with two heap blocks used

memory used by an object that has no fields or an array that has no elements), a minimum free space size can be specified to avoid excessive fragmentation caused by very small free spaces that can't be used. If the free space that would be left over inside a space that is being used for an allocation is smaller than this minimum size then the whole space is allocated even though a small section at the end will not be used. No attempt was made to find a better fit in these cases as the wasted space was considered negligible, although it might become more significant if the situation arose frequently.

Garbage Collection

When the garbage collector is invoked, all the currently reachable objects and arrays must be found. Those that are not currently reachable can then be discarded and their memory made available for reuse. A simple "mark and sweep" algorithm is used for determining the reachable objects. All the objects that are reachable from certain roots are marked. After the marking phase is complete, objects that have not been marked are discarded.

We determined the roots of reachability to be the following: The operand stack and local variables in each stack frame of each thread, the C stack (e.g. in case an object has been created but not yet pushed onto the operand stack), the JNI local and global references and static fields of all loaded classes. If an object is found in any of these places then all the objects that are reachable from that object are marked recursively. Should an object be found that has already been marked then the recursion halts.

A macro is used to perform a rough check to see if potential references are in the heap's range (between its lowest point and highest point in memory). If so, another check is needed to make sure that the reference is actually into one of the heap blocks since the heap may not be contiguous in memory. When checking the operand stack, the local variables and the C stack, there is no indication as to which slots contain data of which type. The bit map is consulted to see if a value could be a valid reference. For fields, the field descriptor in the object's class definition can be examined to see if the field is a reference. If it is then it will either be a valid reference or NULL. If a native method is currently being executed then the JNI local references must be checked, along with the JNI global references.

When all the reachable objects and arrays have been marked, the sweeping phase begins. The bit field in each heap block is checked to find the beginnings of objects and arrays and to see if they have been marked as reachable. If marked, it is unmarked (so that it'll be ready for the next garbage collection) and if not, the space is added to the free list and its bit in the bit map is reset.

Should a newly freed space border on another free space then the two are combined to keep fragmentation down. Compaction of allocated spaces is not used since moving objects means references have to be updated.

3.4.5 Representation of Objects and Arrays

An object comprises a header and a pointer to an array of 32-bit integers to store its field values. The header includes a pointer to the class structure representing the object's type, space for flags and locking information. Information about the object's fields and methods is all stored in the class structure. Memory for the object's fields is allocated directly following its header on the heap.

A thread can get an exclusive lock on an object by claiming its monitor. So an object also needs to store a pointer to a `Thread` object if any thread has claimed its monitor and a counter to keep track of how many times that thread has claimed that monitor. Each object has a list of threads that are waiting on it, along with the size of this wait set.

Since arrays can be treated in the same manner as class instances, they contain all the above information. They also require fields to record the number of elements they hold and the type of the elements.

Since an array of a certain type and number of dimensions has its own class type, new classes are created when new types of arrays are created. If another array of the same type is allocated then it will use the existing class definition. When multi-dimensional arrays are created, a new type is created for each dimension if an appropriate one doesn't already exist. The array types that are created are exactly the same as `java.lang.Object` type except for their name.

3.4.6 Exceptions

When exceptions are thrown due to the `throw` keyword in Java, the Java bytecode instruction `athrow` is used. This expects the object that is being thrown to be on top of the operand stack. Other instructions can throw exceptions in the course of execution. An example is an array index instruction throwing a `NullPointerException` if the array pointer on the operand stack is NULL. When an exception is thrown, an instance of the relevant exception class is created and a reference to it is pushed onto the operand stack. Any object can be thrown, but usually it is only useful to throw exception objects.

The process of catching an exception once it has been thrown is an expensive one. The catch clauses of the currently executing method have to be searched for a clause that matches the exception that has been thrown. Each catch clause specifies the instruction range over which it is valid and specifies the type of exception that it should catch. In order for a match to succeed, the instruction that threw the exception has to be within the clause's instruction range and the exception that was thrown has to be the same as (or a subclass of) the type of exception that is specified by the clause. If the catch type of a catch clause is undefined, then that clause will catch any exception that is thrown inside its instruction range.

If all the current method's catch clauses are examined without the exception being caught, the calling method's clauses are consulted recursively. If we do not catch the exception by the time we have consulted all the stack frames, the thread needs to be stopped.

3.4.7 Thread Management

Threads are handled entirely by the virtual machine itself, with no operating system support. While this can be less efficient, it makes them totally portable. Should performance gains be required on a system that provides its own thread support then the virtual machine threads can be changed to use it. A tick counter is kept to check for thread time-slice completion. This counter is decremented once every instruction, so a thread that executes many complex instructions will end up with more actual processor time than one that executes mostly simple instructions.

The basic thread system is quite simple: there are ten priorities and highest priority threads are always executed first. This means that low priority threads can be excluded from being allowed to run, but this is in keeping with the definition in the Java language specification. Thread functionality includes everything required by the java.lang.Thread class as well as providing for `wait()` and `notify()` in the `java.lang.Object` class.

An array of thread rings is stored. Each ring is a circular linked list, holding all the threads of a particular priority. When a new thread needs to be scheduled, every thread is visited to check its eligibility. Sleeping and waiting threads also need to be visited because they need to be checked to see whether or not they have slept the required time. The highest priority potentially runnable thread is then scheduled for execution. Potentially runnable threads include runnable threads, threads that have been waiting to claim an available monitor and threads that have finished waiting or sleeping. Threads of equal priority are executed in a round-robin fashion.

When a thread needs to claim a monitor on an object, it yields and puts itself into the MONITOR_ENTER state. If a thread's state is MONITOR_ENTER and the monitor is available but there are higher priority threads that can be run, the monitor is not given to that thread. This is done because the threads only need the monitor when they are actually going to run and granting more monitor requests than is necessary might increase the likelihood of a deadlock occurring.

If `notifyAll()` is called, all the threads in the object's wait list are notified. If `notify()` is called, a random thread is picked from the wait list to notify. When a thread is notified, it has to reclaim the monitor of the object upon which it was waiting before it can run so it is set MONITOR_ENTER. If the thread held more than one monitor on that object, it has to be given that many monitors back.

3.4.8 Constant Pool Storage

Each class definition contains a constant pool. The constant pool contains all the constants that will be required by the class for the execution of its methods. These include strings, class references, field references, method references and numeric constants and are all loaded from the compiled Java class file during the class loading procedure. Each constant pool entry has a corresponding tag to denote its type.

Internally, each class's constant pool is stored as an array of 32-bit quantities. Numeric constants of `long` and `double` types are both 64-bit quantities and are stored in consecutive slots in big-endian order. All field, method and class references are composed of references to other constant pool items. These references are all indices into the constant pool and are 16 bits in size. Since at most two of these indices are needed in the same slot, they fit the available 32 bits. At the lowest level, all these references are resolved to strings and it is these strings that are used to find the correct field, method or class. A (32-bit) pointer to the string is stored in the corresponding slot.

A detailed description of the constant pool format is given in [55]. The reason that references are defined in this fashion is to make the classes as portable as possible. This means that a class can reference another class that may not already have been loaded, allowing for dynamic linking of classes, methods and fields.

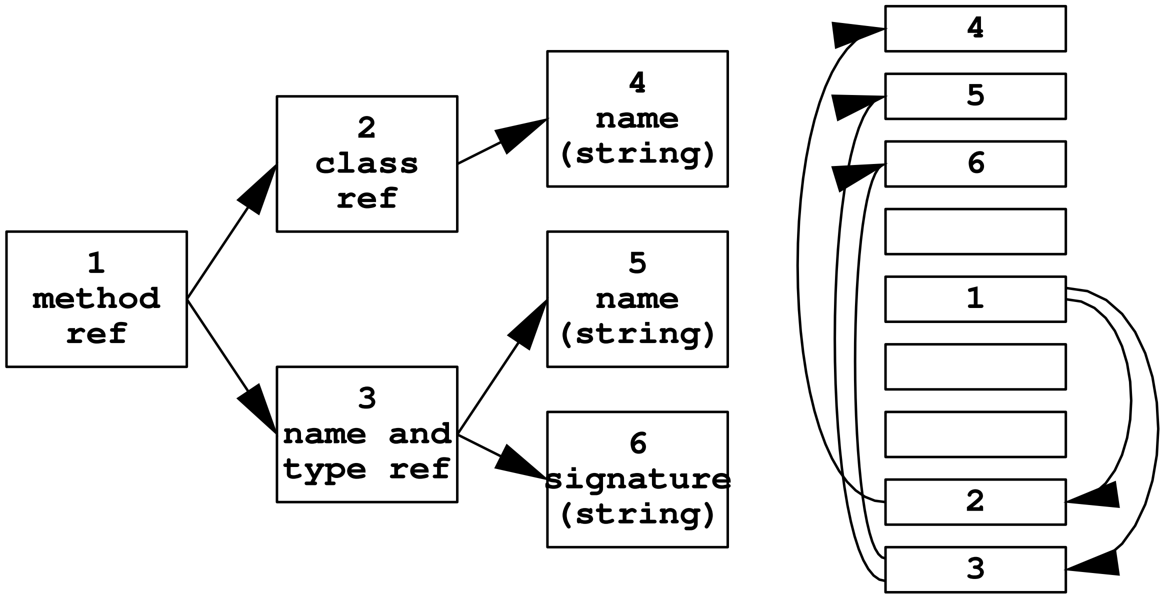

Figure 9: Depiction of the structure of a method reference in a constant pool.

3.4.9 Class Loading

Classes are loaded from a Java class file into the class list when the need arises. The JVM searches the class list before doing so.

Before a class is used it needs to be initialised. This is done just after the class is loaded to avoid having to check that classes are initialised whenever they are used. If a class has an initialisation method (called `<clinit>`) then it has to be executed to initialise the class. Static fields are fields that are common to all instances of a certain class and to the class itself. Space for a class's static variables is allocated at load time. This will not waste memory since classes are only loaded when they are needed.

3.4.10 Hash Table for String Constants

A hash table is used in order to store pointers to strings. Duplicates are avoided; this has two advantages:

- because of the fact that duplicate strings are not allocated memory, space is saved. Many strings in Java classes are duplicated due to the fact that classes and methods are specified by name. An example is that all classes that are directly inherited from the `java.lang.Object` class have the string `‘‘java/lang/Object’’` in their constant pool.
- because identical strings always have the same pointer, the equality of strings can easily be determined by comparing their pointers. This is a gain when searching for classes or methods because these operations require many string equality comparisons.

All internally used strings (e.g. names of classes, names and signatures of methods and fields) are added to the hash table to avoid duplication and maintain consistency.

3.5 Java Native Interface

The Java Native Interface (JNI) is provided so that a standard interface is available for native methods to manipulate Java objects and call Java methods. LEAN implemented the JNI version from the JDK 1.1 release. The JNI functionality is covered in the JDK documentation [54]; we discuss some of the implementation here.

The first parameter that is passed to all native methods and JNI functions is the JNI environment pointer which contains information that is needed or that needs to be updated. This data structure includes a pointer to the array of all the JNI functions that the native methods can call. Because of the way that the parameters are passed to and from the native C methods from the JVM, all native methods must be compiled with the C calling convention.

A native method can raise an exception. When this is done, the exception field of the thread's JNI data structure is set to point to the exception object that was raised. The native code can clear the exception or it can allow it to be handled by the Java interpreter on return from the native method. Exceptions that are thrown in Java methods that are called from native methods are passed back to the native method. If an invocation of a Java method causes an exception to be thrown then the return value of that Java method should be ignored since it will not be valid.

There are three ways to pass parameters to Java methods that are called from native methods. These are by variable arguments using elipsis (...) or by passing a `va_list` [57] or by passing an array of `jvalues` (`jvalue` is a union of all the primitive Java types). In the first case, a `va_list` is created and then it is handled in the same way as the second case. In all three cases, the parameters are put into an array of 32-bit integers that is passed to the function that invokes the interpreter (with `long` and `double` data types taking up two slots).

When a Java object is passed to a native method, a local reference is created for that object. This is used to record the object's existence so that the garbage collector does not throw it away. All JNI functions that return objects or arrays automatically create local references for those objects as well. Local references can be converted to global references which can only be cleared explicitly.

When a native method is called, our JNI structure is retrieved and space is allocated for the local references which are created. Since our thread implementation means that we can only have one thread executing native code at a time, we only really need one of these structures but we keep one per thread.

In LEAN, monitor enter and monitor exit operations cannot be performed from native methods since a thread might have to wait for that monitor to become available. Because we're using interpreter-controlled threads there is no way to make the thread restart in the middle of a native method once it has entered the monitor. To overcome this, the Java method that calls the native method should try to do all necessary locking before it calls the native method.

3.6 Optimisations

Since the virtual machine relies so much on the constant pool data, constant pool access should be optimised as much as possible. Items in the constant pool often refer to other items by an index into the pool. At the lowest level the items pointed to are strings which are used to look up data by searching tables. Items that are referenced in this way are classes, methods and fields. This method of following chains of indices in the constant pool followed by a search is inefficient, so the initial reference must be replaced by a pointer directly to the item required (either a class, method or field descriptor) once it has been found.

This resolution of constant pool references can be done either at load-time or dynamically at run-time. The advantage of load-time resolution is that by the time the code is executed and these items referenced, they have already been resolved and very little work is needed in order to access them. If load-time resolution is performed then every other class that is referenced from the relevant class's constant pool (even if that other class is not actually used) will be loaded when the resolution is performed. LEAN uses dynamic resolution as described below.

3.6.1 Permanent Constant Pool Resolution

Permanent run-time resolution involves overwriting constant pool items with their resolved versions the first time they are used. Items must be flagged to indicated which have been resolved. If a reference is made to an item that has not been resolved then it must be looked up in the usual way. Then the pointer to the retrieved structure must be stored in the constant pool slot and the item must be flagged as resolved. This permanent run-time resolution is the default resolution option.

The next scheme does away with the need for a check on each reference.

3.6.2 `_quick` Instruction Variants

LEAN implements both the `_quick` instructions (an extension of the Java bytecode instruction set) and my own variation on these, which is called `_quick2`. All these instructions are not present in the output of the Java compiler, but are rather produced by the interpreter itself and used to overwrite other instructions. They are effectively another method of doing constant pool resolution.

There is a `_quick` alternative instruction available for each instruction that references the constant pool. When an instruction that accesses the constant pool is executed, it executes in the normal way, doing normal constant pool resolution as described above. Once it has done this, it overwrites itself with its `_quick` version and stores the information that was acquired from the constant pool lookup. The advantage of this is that the next time the interpreter gets to the same point in the program it will encounter the quick instruction instead of the one that was overwritten. The `_quick` instruction that is executed uses the constant pool information that has been stored without having to check that it has been resolved. Of course, the `_quick` instructions must expect the same input from the operand stack as their non-optimised alternatives.

Different instructions store this information in different ways. Some just use the resolved constant pool item as it is, the same as would be done with normal constant pool resolution (except that the `_quick` instruction knows that it has been resolved and doesn't have to check). Others overwrite their operands with the resolved information.

Two different versions of `_quick` instructions were implemented on LEAN. The first is the set of quick instructions suggested in Appendix A of Sun's Java Virtual Machine Specification [55]. The second is our variation on these, which we call `_quick2`. The most interesting `_quick` instructions are those that are used for method invocation and field access and will be discussed here briefly.

The underlying mechanism is the same for reading and writing fields of the same type. In the `_quick` implementation of static field accesses, the actual address of the required static field is stored in the constant pool. In the `_quick2` implementation, the operands are overwritten instead. The first operand is overwritten with the index of the requested field inside the requested class, the second with the index of the requested class in the current class's class table. The class table is only used when the virtual machine is compiled to use `_quick2` instructions; it contains references to all classes used by a particular class. When the requested class is the current class, a different `_quick2` instruction is used that only needs the static field's index, not the class table index. The `_quick` implementation of instance field access stores the index of the requested field in the instruction's first operand byte. The actual object reference is taken off the operand stack as usual. There is no `_quick2` variation on instance field access instructions; the `_quick` instruction is used instead.

With virtual method invocations, the `_quick` version stores the virtual table index of the method in one of the operand bytes and the number of arguments that the method takes in the other operand byte. In the `_quick` implementation of non-virtual and static method invocations, a pointer directly to the relevant internal method structure is stored in the constant pool. These two have `_quick2` variations which store the method index and class table index in the operand bytes. There are also special `_quick2` instructions for these two that are used when the required class is the current class.

3.7 Pre-Loaded Classes

When classes are used on the DSTV decoder, the problem arises that there is no file system from which to load the class files. These classes need to be stored in ROM on the decoder, so they cannot be changed in any way during execution.

3.7.1 Initial solution

The problem was initially solved by writing a utility to convert class files into C arrays that could be linked into the program. No processing was done when converting the classes; all that was done was that arrays of bytes were generated, giving an exact copy of what was in the class file. These generated files were compiled and linked with the machine when it was compiled for the target system. When the classes were needed, they were loaded from memory in the same way that they were loaded from a file system. This obviously wasted space as the classes were present in both their raw and their loaded forms.

3.7.2 Revised solution

A better method was then devised. This required loading the class from the file system as usual and then writing the class information from the internal class descriptor to a file. This file does not just contain a simple array of bytes as in the previous scheme but a C declaration of the class structure with initialisers to make it a copy of the class descriptor in its loaded format. This means that these classes can be used directly by the interpreter without having to be loaded. This eliminates both the speed overhead of having to load the classes and also the space overhead of storing the class in both its raw and its loaded format.

This method also eliminates the second problem - the problem that classes that are stored in ROM cannot be altered. If the classes are optimised before they are written to files, the class structures in the file will already have been optimised. This includes resolving each class's constant pool and overwriting the relevant instructions with their `_quick` variants if desired.

This pre-loading process is done by a separate program. Classes are stored as structures in C source files as described. These source files are then compiled and linked into the interpreter so that the classes can be used.

Because the classes in ROM cannot be altered, they cannot have static variable space allocated. Each class that is loaded is given an index into a global static variable array that is allocated at run time. The same applies to a few other internal variables that need to be modified in each class structure.

Pre-loaded classes can be used in conjunction with normally loaded classes. When a class is required the array of pre-loaded classes is searched first. If the class is not found in this array, the virtual machine attempts to load it another way. Normally this "other way" would be from the file system but on a decoder it would be via satellite or modem.

3.7.3 Modification of string hash tables for ROM storage

If pre-loaded classes are used then all the strings that are needed for those classes will be known by the utility that is generating them. During the pre-loading process they are added to the hash table as usual. When the classes are written out in pre-loaded format, the strings that have been added to the hash table are written to a file as well. The strings are stored in an array, sorted in alphabetical order.

When the virtual machine is running and a new string is added, this array is searched with a binary search to see if the string is there. If it is, a pointer to that string is returned. If it is not found then the string is added to the hash table. This way, the hash table is not needed during execution unless classes that were not pre-loaded are used.

Chapter 4

Language Design

4.1 Adding Persistence to Java

PLaVa was developed by adding persistence to the existing Java language by modifying the virtual machine. PLaVa was designed with the goal of adding *orthogonal persistence* to Java.

The following is a classification of PLaVa according to some of the criteria set out by Moss and Hosking in [19]:

- *Does code persist?* Initially, classes were not made persistent. However, since classes are seen as normal objects in Java by way of being instances of the `Class` class, they need to be made persistent in order to achieve orthogonality. By doing this, code was made persistent as well.

- *Does execution state persist?* In PLaVa, thread execution state is internal to the virtual machine so it can be made persistent. This state is however different from external state such as file descriptors and network connections which present problems when made persistent.

- *Does it use transactions?* PLaVa does not (yet) support transactions. Changes are made permanent globally by calling the `stabiliseAll()` method to checkpoint the store or when the virtual machine finishes execution normally.

- *Is the language changed?* The source language is not changed and neither is the bytecode definition.

An alternative way of adding persistence to Java is to change the compiler (e.g. [7]). We decided not to use this approach so that class files produced by standard compilers can be run without having to be recompiled (especially useful when running the standard libraries).

Although the language design was not specifically based on that used for PJama, common goals have meant that the designs are very similar.

4.2 The PStore class

In PLaVa, access to the persistent store is provided via the `PStore` class. Instances of this class cannot be created explicitly by the user, but the class contains static methods that are used to obtain instances. Only one store may be open at a time because only one copy of each of the supporting data structures is kept. Methods for creating and opening stores via Java calls were initially included but were later dropped when classes were made persistent on the store and the store was specified via a command line option. These methods were:

- `PStore createStore(String storeName)`: This method creates a new persistent store. If a store of that name already exists then a `StoreExistsException` is thrown. An instance of the `PStore` class representing the store is returned.
- `PStore openStore(String storeName)`: This method opens an existing persistent store of the given name. If the store is not found then a `StoreNotFoundException` is thrown. An instance of the `PStore` class representing the store is returned.

These calls have been replaced by:

- `PStore getStoreRoot()`: This method returns a `PStore` object representing the store that was specified on the command line. If the store is a newly-created store then a new `PStore` object is created for it and returned.

When a store has been created or opened, persistent roots can be created, retrieved or removed. Methods that perform these functions can be called on any instance of the `PStore` class.

- `void addRoot(String name, Object ob)`: Adds the specified object as a new persistent root. It is given the specified name so that it can be retrieved at a later stage.
- `Object getRoot(String name)`: Returns the specified root if a root of that name exists. It must be typecast to its actual type.
- `void removeRoot(String Name)`: Removes the named root from the store's list of roots.
- `void stabiliseAll()`: Updates the store by writing all changed objects back to disk and promoting all newly reachable objects.

To convert a Java program to a persistent program, only a couple of changes are needed:

1. Retrieve the root object(s) for the program that you wish to run.
2. Insert calls to stabilise the store at relevant points if needed. This is not essential as the store is stabilised implicitly when the program execution ends.

A short example showing how the store is opened and the objects accessed follows. In practice, the calls for opening the store and accessing roots can throw exceptions which should be caught.

```
PStore p = PStore.getStoreRoot("db.store"); /* Open store "db.store" */

DBroot mydb = (DBroot) p.getRoot("DB1"); /* Get root "DB1" */

/* Run the code in the object you've accessed. All other persistent
   objects that are referenced in the code will automatically be faulted
```

```
   in during execution, so this code may be written in standard Java
   without any thought given to persistence. */
mydb.run();

p.stabiliseAll(); /* write out new/changed objects */
```

4.3 Persistence Independence and Persistence Identification

In order for orthogonal persistence to be added to a language successfully, it must be added with a high degree of persistence independence. This is sometimes called persistence transparency in the literature, which illustrates the effect that we wish to achieve. Only the act of loading or saving persistent roots is not persistence independent in PLaVa, but this constitutes only a few lines of code. Stabilisation calls may also be inserted at points in the program but they are not essential as there is an implicit stabilise at the end of execution. This final stabilise is not done if execution ends abnormally, so if the user wants data to be saved even if the application ends abnormally, stabilise calls must be used.

In our implementation no changes to the Java language are made, allowing existing Java libraries to be used, whether in Java source or compiled classfile format. Specifically, all of the standard Java libraries can be used with no problem.

As has been mentioned, persistence identification is done by registering named roots with the store and any objects reachable from these roots will automatically become persistent. At the implementation level, there is only one root for the store and all the named roots are reachable from it, thus automatically becoming persistent. Some security measures can be built into the process of adding and deleting roots, but this has not been examined in detail.

4.4 Data Type Orthogonality

Data of all types can be made persistent; all arrays and instances of any class that are reachable from persistent roots are made persistent. If any of this data is used for representing state external to the virtual machine then it will be saved with its current value but this will in general not be valid if it is re-used during a subsequent execution. The problems with external state have been given a lot of attention [4, 20, 21, 25] but no satisfactory solution has yet presented itself.

4.4.1 Threads

Due to the nature of PLaVa's thread implementation (unmodified since the version of LEAN presented in chapter 3), all information used by threads is internal to the virtual machine. This means that we can make any execution state that is encapsulated by a `Thread` object persistent. However, there are reasons why we might not want to do this as well as a number of implementation hurdles to be overcome.

Issues with Persistent Threads

Due to the way that threads are handled in non-persistent Java, making threads persist may be undesirable. It will introduce code that is not persistence independent as persistent threads will require special handling in order to be *suspended* during or at the end of an execution and *resumed* during a subsequent execution. In the case of PJama which models the store on the idea of a suspended virtual machine [21], persistent threads can be used to implement long-running daemon threads for tasks such as garbage collection.

If a thread which holds locks on objects is made persistent, those locks must be saved as well and the specified objects will not be available to other threads. However, this remains the programmer's problem and the consequences are similar to those in the non-persistent case. The difference is that an application might require the use of objects that were locked by a persistent thread in a previous application and are still locked. Resolving this problem depends on the model that is chosen for persistent threads.

Chapter 5

Direct Addressing ($PLaVa_1$)

5.1 Aim

The first version of PLaVa (called $PLaVa_1$) aimed at introducing persistence while minimising changes to the original portable lightweight machine. When this was working, the system was further changed to add more functionality and to investigate different memory management approaches, so $PLaVa_2$ was produced. This chapter covers the $PLaVa_1$ design and implementation.

5.2 Changes to Memory Management

Most of the memory management in $PLaVa_1$ is the same as it was in the non-persistent LEAN machine. One major difference is that the heap organisation has been changed only to use one heap block rather than a chain of blocks such as was used in LEAN.

In order to reference persistent objects, persistent identifers (PIDs) were introduced. PIDs are 32 bits in length, which makes them the same size as memory addresses and allows references in both formats to be easily interchangable. PIDs are distinguishable from memory addresses because their sign bits are set and checks are done in software. The `ISPID()` macro returns true if the reference passed to it is a PID.

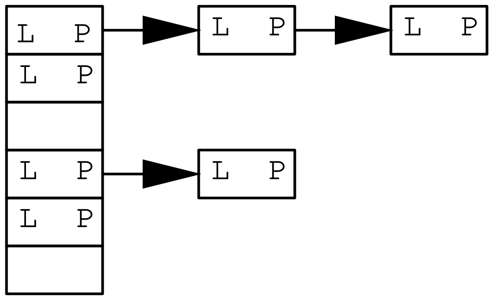

Figure 10: Depiction of RPOT organisation

In PLaVa all objects are kept together in the GC heap, since a separate cache for persistent objects will consume more memory space, especially when promoting objects. The only additional structures introduced are those that record the relationship between disk and memory copies of data.

When an object is faulted in, its PID to local address (LA) mapping is added to a resident persistent object table (RPOT) so that future residency checks will find it. This table is a hash table addressable by PID. Like the environment stacks it is allocated from system memory separately from the GC heap. The RPOT does not need to be addressable by LA as well, because an object's PID is stored in the object header in order to facilitate deswizzling. This also provides an easy way to tell that an object does not exist in the persistent store (it has a null PID in its header).

The marking phase of garbage collection is modified to mark objects that are reachable from the RPOT as well as those reachable from the virtual machine's stacks, etc. The sweep phase is left unchanged.

5.3 Faulting and Swizzling

An object is faulted into the heap when a reference to it is loaded onto the stack. This means that any instruction that pops the reference off the stack to use it can assume that the object has already been faulted in. There are only two instructions that need to be modified to ensure that we fault in all the required objects. These are:

- `getfield`: This instruction is used for loading the contents of an object's field onto the operand stack. References are identified from the field's signature. Its signature starts with '`L`' in the case of a reference to a class instance or with a '[' in the case of a reference to an array.

- `aaload`: This instruction is used for loading a reference onto the operand stack from an array of references.

If references are pushed onto the stack simply to copy them to another location then faulting in the objects that they reference is wasteful, but this should not happen often. Measurements such as those in [58, 59, 60] need to be done to see if this assumption is correct.

Because all parameters that are passed to methods are taken from the operand stack, all references that are passed (including the target object) reference resident objects. This automatically satisfies target object residency and formal parameter residency as discussed in [36, 37]. Procedure result residency will also automatically be enforced as the instruction that returns references from methods takes the return value from the method's operand stack.

In compiled languages considered in [33], actually discovering the original location of a reference so that it can be swizzled can be problematic. In PLaVa this is simplified by the fact that discoveries can be isolated to the execution of two bytecode instructions that actually fault the objects in. The PID is swizzled at its source so that future uses encounter the LA directly. Lazy, in-place swizzling is used.

Our implementation of the Java Native Interface (JNI) [54] is altered to swizzle references that are requested for use in native methods (and to fault in objects as required). The JNI provides calls to access fields of objects and elements of arrays of different types. E.g., if the user wanted to read from an `int` field from native code, he would use the `GetIntField()` call. The functions that implement these calls are modified in the same way as the code implementing the analogous bytecode instructions.

5.4 Updating

All objects that are changed need to be marked as such so that the store can be updated during a stabilise operation. Bytecodes that write to fields of objects or elements of arrays also set their `Dirty` flag. If a reference field in an object, or an element in an array of references, is changed then a separate `RefsDirty` flag is set. These flags are stored in the object header along with other flags used by the original JVM. The two separate flags make the process of promoting changed objects more efficient. If none of an object's reference fields have been updated its fields do not have to be checked and it can simply be written to the store. Instructions affected are:

- `putfield`: This instruction needs to flag the changed class instance as updated. It is marked with the `RefsDirty` flag if it is a reference field that is updated (determined by looking at the field's signature). In the implementation of `putfield`, the signature has to be checked to see if fields are 32 or 64 bits wide, so this is not an extra expense.
- *x*`astore` family: These instructions need to flag the target array as updated (with the `RefsDirty` flag in the case of `aastore`).

Our JNI implementation is altered to flag objects changed in native methods. All calls for writing to objects set the `Dirty` flag and those that modify reference fields set the `RefsDirty` flag. E.g., if the user called `SetObjectField()` then the implementation would set `RefsDirty`.

When stabilising, we must save all dirty objects (i.e. objects that have been changed since the previous stabilise) and all new objects that have become (transitively) reachable from persistent objects.

Stabilising is simple in the PLaVa$_1$ context, since all reachable objects are in the heap. All objects transitively reachable from objects in the RPOT that are either dirty or new. This is done recursively, by visiting child objects and saving them before saving the current object. By doing this, we obtain PIDs for the child objects so that they can be saved. These newly persistent objects have RPOT entries made for them.

Objects that are marked as dirty but not as having dirty references do not need to have their children checked, but swizzled references still need to be deswizzled for purposes of saving.

PLaVa$_1$ used a simple store implementation in which all new objects were simply appended to the existing data in the order in which they were promoted.

5.5 Catering for Alternative Mechanisms in PLaVa$_1$

This section presents the alternatives that have been implemented for comparative purposes in addition to the "standard" configuration of the machine. Parameters can be set to select different flavours of swizzling, updating and RPOT lookup.

5.5.1 Swizzling Alternatives

Swizzling can be turned off. In this case, PIDs in objects are never overwritten with LAs and a residency check needs to be done every time such a reference is used. If few fields are dereferenced more than once then there will be a performance gain from no swizzling.

The garbage collector regards objects that are only reachable via PIDs as unreachable, which is a significant number of objects when no swizzling is used. A variant of the no-swizzling option requires that the garbage collector do residency checks on PIDs and mark the objects if they are resident. (This technique can be used when swizzling is in effect but it doesn't has as much effect in that case.)

When objects are updated in the store, their swizzled fields are deswizzled back to PID form. This means that these fields are likely to cause false object faults thereafter. The previous values of all reference fields can be recorded in order to restore them to their LA values after the write-back. Reswizzling is thus another option that can be enabled or disabled.

To summarise, we have two main swizzling modes: no swizzling and swizzling references on use. For the latter, references can optionally be reswizzled after they have been deswizzled for a stabilise. For both modes, marking during garbage collection can optionally include residency checks on references in PID form.

5.5.2 Update Tracking Alternatives

When an object needs to have its reference fields checked to detect newly persistent objects, all field signatures need to be examined. Reference fields could be more efficiently identified by clustering them within objects. Alternatively, a bitmap could track updates to fields and/or provide quick differentiation between reference and other fields. The bitmap update-tracking method will require more space and extra processing during execution but fewer checks during a stabilise. These alternatives are worth exploring but are not currently implemented.

5.5.3 Reverse RPOT Lookups

Instead of storing an object's PID in its header, the RPOT can be queried by LA in order to retrieve the PID. This is thus another machine configuration parameter.

5.6 Conclusion

PLaVa$_1$ ran simple persistent programs including a CD cataloging application and an application that runs queries on a map of the London Underground. However, it had the restriction that all reachable objects (both persistent and transient) must be able to fit in the heap at all points in the program. To overcome this, indirect addressing was introduced and PLaVa$_2$ was created, which is the subject of the next chapter.

Chapter 6

Indirect Addressing (PLaVa$_2$)

A second version of PLaVa (PLaVa$_2$) was built, which used indirect addressing via handles to allow for a compacting heap garbage collector. The store interface was also changed to conform to the PJSL specifications[61]. Eviction of reachable, possibly transient objects was incorporated to prevent programs being terminated due to lack of heap space.

The chapter starts with an outline of the changes that were made to the heap management and the RPOT to support PLaVa$_2$ functionality. The biggest changes in PLaVa$_2$ are in the way garbage collection works and this is covered in the following sections along with the processes of swapping objects in and out of memory and the process of stabilising. The issue of storing classes on the store is handled next, followed by experiences with interfacing with the PJSL store. The chapter ends with a discussion of different configurations that can be used in PLaVa$_2$.

6.1 Modifications to the Heap

The original garbage collector was replaced with a compacting garbage collector. If objects are allowed to move and references are direct pointers, then all the pointers to a moved object need to be found and changed to point to its new location. To avoid this, we use indirection. We have taken care to ensure that direct pointers do not exist on the native stack or in the interpreter. Our JNI implementation also ensures

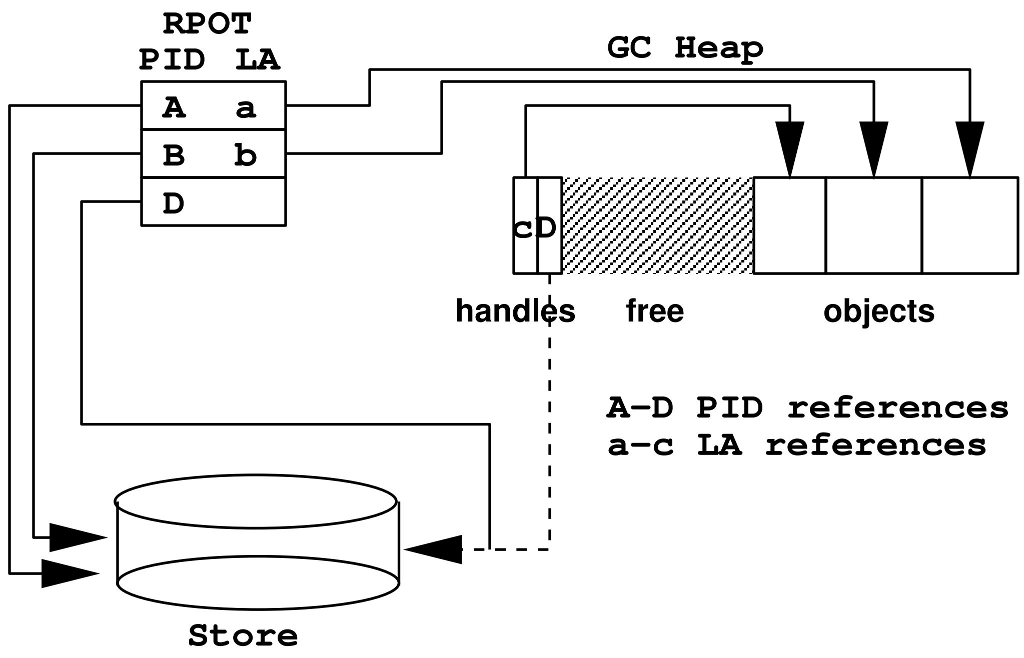

Figure 11: Local addresses are kept in the RPOT or handle table. The rightmost handle references an object that has been swapped out to the store.

that no direct pointers into the heap can arise.

Handles reside in the GC heap, but are allocated starting at the top of the heap, whereas objects are allocated starting at the bottom. This separation is useful because the handles must remain fixed and would cause immovable islands in the heap, hence making compaction troublesome if they resided with the objects. Also, the fact that handles are of fixed size allows them to be stored more efficiently if they are collected in one place because headers can be omitted. Since a compacting garbage collection algorithm is used and objects are never discarded outside a garbage collection session, the GC heap is always divided into the handle table, an area of free memory and an area of contiguous objects (see figure 11).

In order to save space, we use the LA field of an object's RPOT entry as its handle if it was loaded from the store. This required changes to the RPOT implementation as discussed in the next section.

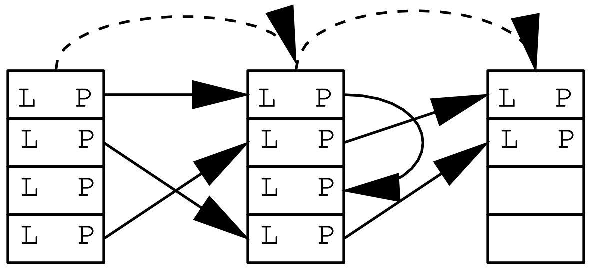

Figure 12: New block-allocated RPOT

6.2 Changes to RPOT

To discover whether an arbitrary field is a handle, we have to check if it is within the area of memory reserved for handles or if it is within the area(s) of memory used for the RPOT. This situation will arise when checking the operand stacks or the C stack where we cannot check field signatures.

To facilitate this, we changed the implementation of the RPOT to allocate space for overflow entries in a pre-allocated block of memory. Should this block be full when a new entry is required, another block is allocated. These blocks are chained together in a linked list so that they can be scanned quickly to determine whether or not an address falls within any of their ranges.

6.3 Garbage Collection

If there is not enough heap space to satisfy an allocation request, the garbage collector can get rid of objects that are no longer needed. If this would not free up enough space then some other action needs to be taken, e.g. expanding the heap if possible or swapping reachable objects to the store.

In PLaVa$_2$, the marking phase of a garbage collection operation is extended to incorporate objects reachable from the machine, the RPOT, and objects that are reachable only from objects that have been swapped out. The sweep phase discards all objects that were not marked and performs compaction of the remaining objects in the heap. If an object is moved, its position is altered in the bitmap and the contents of its handle or RPOT entry are updated with its new memory address. If an object is swapped out, its position is cleared in the bitmap and the contents of its handle or RPOT entry are changed to contain its PID instead.

6.4 Categorising Objects

The garbage collection process starts in the conventional way by marking reachable objects. Objects that are reachable from the machine (i.e., via operand stacks, references from other reachable objects, static references from classes and local or global native method references) are marked with one flag and objects that are reachable from the RPOT are marked with another. We call these objects "reachable" and "RPOT-reachable" respectively. Of course, a number of objects are marked with both flags.

Using these flags as well as the "dirty" flag (that tells us whether or not an object has been changed since it was last saved) and the fact that the object might or might not be persistent, we can categorise all the objects in memory according to the overhead that evicting them will incur.

We have two goals when selecting objects to swap out. One is to choose objects apparently unlikely to be used in future and the other is to select objects that will result in the least eviction overhead. The latter goal is the more critical in our case, but the former is also taken into consideration.

We define a *pseudo-persistent* object as one that was created after the last stabilise and has since become reachable from the RPOT. For eviction purposes we identify eight different categories of object, listed here (using the abbreviations in table 6.1) in order of preference for swapping out:

0. *UT* These objects are unusable and are discarded; they are the objects that would be discarded in a standard garbage collection procedure.

1. *UCP*, UCp_1 Both exist on the store already and require no writes because they are clean. In the UCP case, the objects have their RPOT entries removed since they are unreachable and if they are needed again it'll be the same as a normal object fault.

U	unreachable
R	reachable
C	clean
D	dirty
T	transient
P	persistent
p_0	pseudo-persistent and never swapped before
p_1	pseudo-persistent and been swapped before

Table 1: Abbreviations used in object categorisation.

2. *UDP*, UDp_1 Both exist on the store already but require updates because they are dirty. The rest of their swapping out procedure is the same as that for category 1.

3. Up_0 These objects have not been swapped before, so they are written to the store.

4. *RCP*, RCp_1 The same as category 1, except that being reachable, they are more likely to be swapped back in for reuse, making it less efficient to swap them out.

5. *RDP*, RDp_1 The same as category 2, except that being reachable, they are more likely to be swapped back in for reuse, making it less efficient to swap them out.

6. Rp_0 The same as category 3, except that being reachable, they are more likely to be swapped back in for reuse, making it less efficient to swap them out.

7. *RT* These objects are the least favourable to swap out because they are unlikely to become persistent (at least, that is the way it appears at the time of marking). It is therefore wasteful to swap them out because we are unlikely to gain future advantage from doing so.

Each time a new object is found during the marking process, it is categorised according to the above criteria. Reachable objects are marked first, followed by RPOT-reachable. An object marked in the first (machine-reachability) pass that is marked again in the second (RPOT-reachability) pass will move from one category to another. A running total is kept for each category, storing the number of words occupied by all the resident objects in that category.

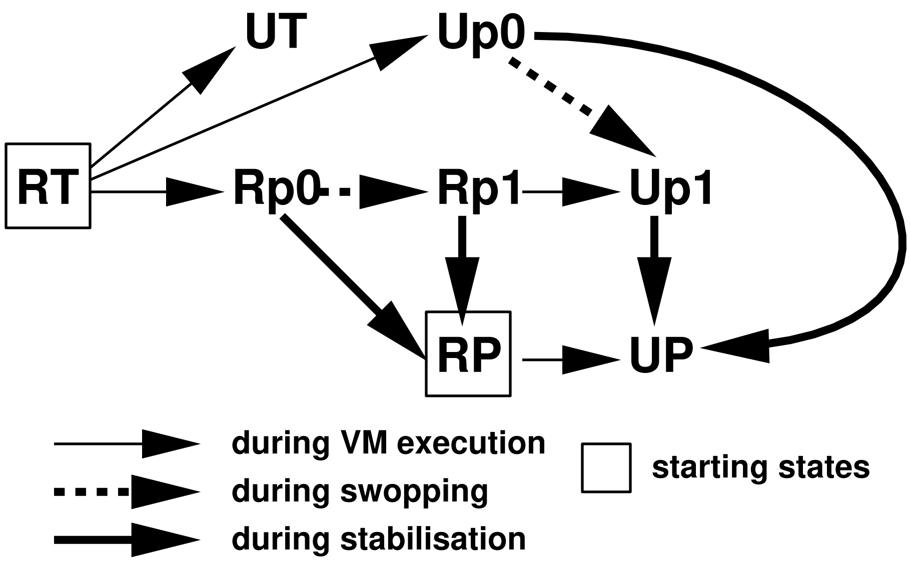

Figure 13: The life cycle of a PLaVa object, displaying how it moves between different states. Adding dirty/clean states to this allows us to divide them up into the eight categories.

The "lifecycle" of a PLaVa object is shown in figure 13, demonstrating how it moves between the different categories during its lifetime. Some transitions happen during normal execution (e.g., becoming unreachable), others happen during swapping and others happen only during a stabilise. No distinction is made between dirty and clean objects in the diagram, but they should be easy to envisage.

6.5 Selecting Categories for Eviction

In the categories that we've defined, low eviction overhead is considered a bigger factor in selecting objects to swap out than probability of re-use.

Once objects have been categorised, we decide which ones we need to swap out. The amount of memory that we need to reclaim is defined by the "eviction factor". This defines what fraction of the heap we want to reclaim during garbage collection. The higher this number, the more memory we'll have for objects in future. However, there are two possible drawbacks. The first is the obvious problem that because we have swapped out many objects, we are more likely to have swapped out objects that we'll need to use again. The other is that freeing up more memory is likely to require swapping out objects from categories with higher eviction overhead. The eviction factor is a machine configuration parameter that gives some control over the amount of eviction.

Once we have determined how much memory we have to reclaim, we calculate how many categories we need to evict by considering the size of each category.

6.5.1 Fine-Grained Selection

Because of the coarse granularity of the categories, we could end up swapping out many more objects than we need to if we use an "all or nothing" approach for each category. To overcome this, we can choose specific objects from within the last required category in order to avoid swapping out too many objects. This finer-grained selection can be done with any standard replacement policy (e.g., FIFO, LRU, Enhanced Second Chance [62], etc.). This strategy is especially useful when the last required category is a category that includes reachable and/or dirty objects.

When eviction of reachable objects (categories 4 to 7) is needed, it is possible that a hybrid scheme would be preferable to one which always emphasises eviction overhead above usage statistics. In addition, purely scalar objects are better to swap out than objects containing references.

6.5.2 Stack-Directed Marking

Since an object can only be used if a reference to it is pushed onto the operand stack, we can consider all objects referenced from stack frames near the top of the stack to be useful in that they are likely to be used soon. This information is readily available to us and, unlike most standard replacement policies, we do not have to introduce time stamps or any other way of deciding which objects are important. We can use this area at the top of the stack to mark objects similarly to the process described in [38].

To take advantage of this, we modified the marking stage of garbage collection in PLaVa$_2$ to flag objects reachable from the top stack frames. We can define the

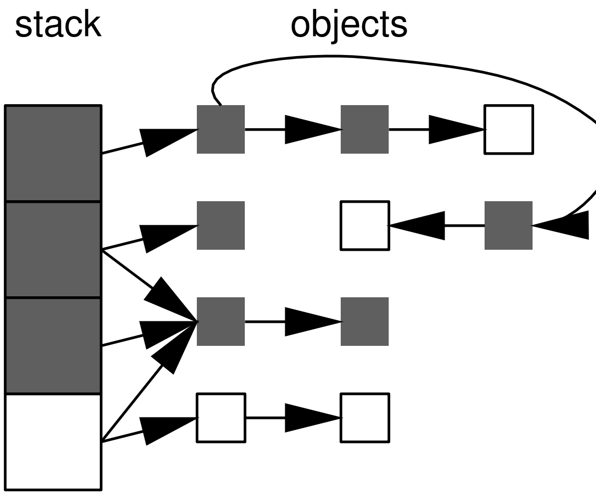

Figure 14: Depiction of stack-directed marking

number of stack frames from the top of the stack to consider important. We can also select the number of references to follow from the stack while finding objects to keep. This is useful in the case of objects with embedded data, e.g. `String` objects contain an array of `char`s where it would be a good idea to keep this actual character data as well as the wrapping `String` object. See figure 14 for a depiction of the affect of these two machine configuration parameters.

In order to be sure that enough space is reclaimed, the sizes of the eviction categories are modified not to include the objects marked in this way. Thus, when the category sizes are considered to calculate which categories to evict, these objects that are exempt from eviction will not be counted among the potential victims.

Because different running threads have different object usage characteristics, we will gain advantage by using run-time information to determine the number of stack frames to mark in this way independently for each thread. This is the goal of other current research [38].

6.6 Process of Swapping Out

Each object in the heap is examined to find its category and hence determine it if must be evicted. If an object has no category marking then it is unreachable so it can be discarded as would be the case with standard garbage collection.

When an object is evicted, the contents of its handle are overwritten with its PID to indicate that it has been swapped out. Persistent objects use the LA field of their RPOT entry as a handle, so that is overwritten in the case of persistent objects.

PID to LA mappings must be kept for transient objects that are swapped out and for children of swapped-out objects because these children have PIDs allocated to them. These mappings are stored in the Temporary Persistent Object Table (TPOT), which is discussed in the next section.

`Thread` objects are not swapped out, as these contain information that is needed during the garbage collection process and must be available for use.

6.6.1 The TPOT

When non-persistent objects are swapped to the store, they are allocated PIDs just like persistent objects are. A table similar to the RPOT maintains their PID to LA mappings. This table is called the Temporary Persistent Object Table (TPOT). The RPOT contains only mappings for persistent objects, whereas the TPOT contains mappings for all non-persistent objects that have been allocated PIDs.

TPOT objects are either non-persistent objects that have been swapped out to free up memory space, or non-persistent objects that are directly referenced by evicted objects (so-called *hidden* objects).

Marking in a garbage collection operation stops if it encounters a reference to an object that has been swapped out. This means that objects may not be found because they are "hidden" behind an object that has been swapped out. Since these hidden objects will all have had TPOT entries created when their parent was swapped out, they can be located via the TPOT. There are thus three marking passes - objects reachable from the machine, from the RPOT and from the TPOT.

If TPOT objects become persistent during a stabilise, their entries are moved from the TPOT to the RPOT.

While the RPOT has been changed to store all of its entries (i.e. leaves) in blocks (i.e. bags) to facilitate determining whether or not a pointer's destination is within the RPOT, the TPOT does not need to offer the same sort of functionality so it has not been changed. This means that the TPOT is still used with leaves, not bags.

6.6.2 Swapping Out in Memory Format

A drawback of swapping out objects is that any references that they contain must be in PID format so that they reference valid objects in the store. In the case of transient (at the time of swapping) objects, this results in allocation of PIDs for objects that might never become persistent, potentially wasting time and space.

If an object's header contains H words and it has S words of scalar data and R reference fields, then it occupies $H+S+R$ words in the heap. A TPOT entry occupies 3 words of auxiliary memory, so if we make a TPOT entry for each reference field in the object we free $H+S-2R$ words of memory. If this becomes negative (i.e. if R is too large) then we are in fact wasting memory by swapping the object out in store format. This could be the case e.g. with arrays of references or internal nodes of a tree with a large fan-out.

To counter this problem, we can swap objects out in memory format. To do this, we keep the object's references as handles. These handles will be valid when the object is swapped back in, but if the object becomes persistent during a stabilise operation then it will have to be loaded into memory and saved again in store format.

If an object is swapped out in memory format we keep embedded references in handle format. Because of this, TPOT entries will not have been created for its children so its children might not be identified as reachable. To get around this, if we swap out an object in memory format, we must inspect its children to see if they exist on the store. If they do not, then their "external reference count" (ERC) is incremented to indicate that they are not useless objects. When a parent of such an object is swapped in, the object's ERC is decremented. The ERC is stored in the area of the header usually used for an object's PID, since objects that require an ERC have not been allocated a PID. A total of the sizes of all objects with ERCs is maintained for use in category selection during garbage collection.

In this way, PLaVa$_2$ can optionally be configured to swap out transient objects in memory format (instead of creating TPOT entries for referenced objects).

6.6.3 Swapping Out with a No-Steal Store

If a store does not support writes of dirty persistent objects outside of a stabilise (i.e. a no-steal store) then temporary copies need to be made of any dirty persistent objects that are swapped out. In this case, the dirty persistent categories (2 and 5) will move lower down the preference list to between the current categories 6 and 7.

When using such a store, each RPOT entry contains an extra field. This holds a PID for the object's temporary copy (if one has been made) so that this copy can replace the original object on a stabilise or if used again by the VM. These temporary objects can be put into a garbage partition in a partitioned store like PJSL so that they can be cleared away easily by the store garbage collector.

6.7 Process of Swapping In

If an object that has been swapped out is required again thereafter then it must be swapped back in. Swapped-out objects are identified by the fact that the contents of their handle is in PID format (or the LA field of their RPOT entry is a PID). This PID is used to identify the object that must be swapped in. The swapping-in process is similar to the process that faults objects in initially, except that the object's old handle is used instead of allocating a new one.

6.8 Stabilising

During a stabilise, we must make sure that all the objects that are reachable from the store have their current state saved to the store. We try to keep this process as simple as possible without saving unnecessary objects. Initially we just saved all objects in the heap that were dirty, but this meant writing some transient objects that were unreachable from persistent roots.

To determine which objects need writing, we mark all objects that are reachable from the RPOT and also all objects that are reachable only from the TPOT. With this information, we can tell which objects are reachable from persistent objects or from objects which have been swapped out. Objects with either of these markings need to be written to the store.

We find each object in the heap by examining the heap bitmap, the same as is done when garbage collecting. When we find an object with one of the above- mentioned markings, we check to see if it is already on the store by checking to see if it has been allocated a PID. If it has a PID and is clean then we do not need to take any action, but if it is dirty then we need to write it. If it has no PID then it must have one allocated and be written.

This process will write children of swapped-out objects because they'll be found when marking objects reachable from the TPOT. In order to identify only objects reachable from the store we would have to swap in objects that have been swapped out. Since memory is a precious resource for us, we opted for this approach which will promote some unnecesary objects (they will be discarded by the store's garbage collector) rather than one that requires us to swap objects back into memory.

During stabilisation, all objects that have become persistent have entries added to the RPOT. Objects that have been swapped out but were not persistent before the current stabilise have their TPOT entries moved to the RPOT to reflect the fact that they have become "properly" persistent.

If objects with RPOT entries are no longer reachable from the machine, their RPOT entries are removed to reclaim space. Handles of unreachable objects are also freed. If we do a large amount of such "cleaning up" during a stabilise, it can become a heavy-weight operation which will affect performance if too many stabilises are done during execution. The effect will depend on individual applications, particularly on whether or not a significant number of the objects that are saved during a stabilise are used again afterwards. Hence this clean-up-on-stabilise can optionally be deselected.

6.8.1 Process of Writing in Store Format

When objects are saved, all references that they contain must be saved as PIDs so that referential integrity will be maintained when they are reloaded (which could be during the same application execution or at any other point in the future).

Firstly, we check to see whether or not the class definition of the object's class has been saved. If it has not, it is saved (this process is discussed in a later section). The class definition's PID is recorded in the object.

If the object is either a class instance or an array of references (i.e. either an array of objects or an array of arrays) then PIDs must be found for each of the references in the object. These PIDs can be obtained in one of four ways:

- The reference is already in PID form so it is left as it is.
- The reference is a handle that contains a PID (i.e. it refers to an object that has been swapped out.
- The reference is a handle that refers to a resident object that has been assigned a PID (i.e. a persistent object or an object that has been swapped out and subsequently swapped back in).
- The reference is a handle that refers to a resident object that has no PID assigned to it. In this case, we need to allocate a PID for the object in question.

6.8.2 Stabilising with Memory-Format Swapping

If all objects are evicted in store format then objects that are not swapped back in again will have had their latest state saved. If we have swapped some objects out in memory format, then these objects must be reloaded and saved in store format. Their TPOT entries will have been flagged to indicate this neccessity.

6.8.3 Stabilising with No-Steal

As was mentioned previously, if we are using a no-steal policy then we will have made temporary copies of dirty persistent objects that had to be evicted. All objects whose RPOT entries contain a non-null "temporary PID" field are thus replaced by their temporary copies during stabilisation.

6.8.4 Undoing Changes

When a stabilise is called, objects that have been written to the store during swapping can be left as they are. If the machine terminates abnormally then we need to undo the changes that have been made to these objects. In order to achieve this, we assign an extra "temporary" PID to objects to hold a temporary copy of it for the purposes of swapping. On a stabilise, objects are replaced by their temporary copies. If an abort is needed then we leave the objects as they are, since they are still in the state in which they were prior to being swapped (i.e. since the previous stabilise).

6.9 Storing Classes

Storing class definitions on the persistent store has the advantage that the store becomes self-contained. It does introduce more complexity in the virtual machine because classes must be faulted in when they are required. Also, the granularity of class loading must be decided, i.e. whether to load the whole class definition when it is used or only to load the core of the definition and to load the bytecodes for each method on demand.

When storing classes on the store, the issue of class evolution arises. This is a non-trivial problem and has not yet been tackled in PLaVa. The problem is discussed in [22] which presents an off-line class evolution system for PJama and [24] which presents the JavaSPIN solution and contains a lot of information about what changes can be made to classes without introducing errors.

6.9.1 First Attempts

Classes were not made persistent in PLaVa$_1$. When an object was saved, only the name of its class was saved in the store. When the object was loaded, the name of its class was loaded so that its class could be retrieved from the file system in the standard Java manner (if it wasn't already loaded).

In PLaVa$_2$ this was extended to store classes on the store in their original classfile format. Thus, when an object is loaded its class is loaded in much the same way as in standard Java, except that the classfile data is taken from the store instead of from the file system. This approach does achieve the goal of storing class data on the persistent store, but it requires that the classfile data be parsed and verified each time it is loaded, as done by a standard Java classloader.

In both these methods static variables are not persistent. In the second method, even though the class is on the store in some form, its static variables are not saved. Even if they were saved, the class needs to be initialised when loaded and this will write initial data to its static variables. When classes are promoted in JavaSPIN [24] this scheme of storing raw classfile data in the store is used, but static variables are stored as well.

6.9.2 Final Version

A subsequent improvement was made to save classes in their loaded format so that they could be faulted in and used without any extra processing. The format in which they're saved is dictated to some degree by the format of loaded classes in the original JVM Wombat. In the Wombat format, all method descriptors (excluding actual code) are allocated in a single block and all field descriptors are allocated in a single block. In the PLaVa format, these descriptors are saved separately so that they can be individually addressed and separately loaded.

Saving Classes

When a class is to be saved, all items in its constant pool must be saved. If items have been resolved, they are saved in their resolved format and if they have not, they are saved in their raw constant pool format. All strings in the constant pool are saved separately and their PIDs are saved in their constant pool indices. Any method reference or field reference in the constant pool that has been resolved is saved separately and its constant pool entry replaced by a PID. Any resolved class references are replaced by the PID of the saved class. All other constant pool entries do not get resolved in the course of virtual machine execution and are saved as they are.

A flag is needed for marking constant pool items that were saved in resolved form.

Previously, items were marked as either resolved or unresolved so that the virtual machine can resolve them only if needed. Now, when we save a class to the store, we need to know whether items in its constant pool were already resolved when it was loaded or have been resolved since loading. In the latter case, the newly resolved items must be written in their resolved form (and flagged as such) but in the former case they have already been written.

The first time a class is saved, these checks are not needed as none of the constant pool items have yet been saved as either resolved or unresolved. It is on updates that these checks need to be done.

When an object is saved to the store, its class must also be saved if it is not already on the store. Storing this class might require that other classes be saved as well. If the object has a field of a class `C` then class `C` must be saved. If the constant pool contains a resolved method reference to a method in another class, that class must be saved as well. In this way, all classes that are reachable via resolved references in the constant pool will be saved.

Faulting in Saved Classes

When a class is needed, its definition is faulted in from the store. This does not include faulting in method bytecodes. Some initialisation must be done to the class when it is faulted in. This includes faulting in its static variables, initialising its virtual method table and initialising its native method mappings. For efficiency, all the strings in the class's constant pool are also faulted when the class is faulted in.

When a saved method is invoked, its PID is used to fault in its method definition. When the method definition is faulted in, the method's bytecode is faulted in. The method's defining class is faulted in as well, because the method will require the class's constant pool data in order to run.

6.9.3 Persistent Threads

In order for a thread to become persistent, its associated environment stack must be saved as well. This requires converting all references on the stack (i.e. references in local variables, on the operand stack and in a few virtual machine-specific fields) to PIDs and promoting non-persistent objects that are reachable from the stack. During execution PLaVa assumes that all references on the stack are in LA form, so either the way that PLaVa checks for PIDs must be changed or all PIDs on the stack must be swizzled when a persistent thread is faulted in. The latter is the same as eager swizzling normal objects, so it's preferable to changing the virtual machine.

PLaVa threads cannot fully represent execution state when they are executing a native method as native methods run outside the control of the virtual machine. So in order to support persistent threads, a stabilise will have to be disallowed while a native method is being executed. The situation is simplified because PLaVa's thread model only allows one native method to be executing at a time and this native method is always the current method (native method invocations are treated as single VM instructions). Therefore it is sufficient just to disallow stabilise calls from a native method.

6.10 Using PJSL

PJSL (Persistent Java Store Layer) [8, 61] is part of the Sphere library being developed for use with the PJama project. Using PJSL with PLaVa achieves two goals. The first is to demonstrate PJSL's ability to work with systems other than PJama and the second is to demonstrate PLaVa's ability to interface with a store other than the simple one developed specially for testing.

6.10.1 Integrating PJSL with PLaVa

Before the PJSL store itself was available, we adapted our existing store to use the PJSL API that was available at that time. Most of the work done to achieve this required wrapping our store's calls with PJSL calls as the functionality of our simple store was a subset of the PJSL functionality. When we subsequently received PJSL, the API had been modified slightly (e.g. history capabilities were in the process of being added (but not working in the release we used), so each call needed a history ID).

PJSL uses a partitioned store model, where each partition has a particular *regime* which is managed in a particular way. Different *kinds* of objects are put into suitable regimes so that they can be handled more efficiently [8, 61]. The PJSL API for invoking operations on objects is arranged in a two-dimensional dispatch array, indexed by the object's kind and the regime in which it resides. The kind-specific operations that we use are:

- `allocate` Used to allocate (but not write) an object.
- `firstWrite` Must be called to write an object the first time that it is written. No other operations must be carried out on the object between the time it is allocated and the time it has this operation called on it.
- `updateAll` We use this operation to update dirty objects. Other operations for updating only portions of objects are also available but these are not used by PLaVa.
- `fetch` Used to retrieve an object into memory.

To map an object of a certain kind to a suitable regime so that it can be stored, `pjsl_map_kind_to_regime` is used. Its parameters are the object's kind and its size. To discover the regime, kind and size of an object already on the store, `pjsl_objInspect` is used. This needs to be called on an object before it can be fetched from the store.

Implementations of operations for a number of kinds are supplied with the PJSL distribution. Most of their names are self-explanatory: Instance Descriptor Kind, Small Instance Kind, Small-Object Napier Kind, Small Scalar Kind, Small Reference Kind, Large Scalar Kind, Large Reference Kind and Large Instance Kind.

Small kinds are those whose objects are all smaller than a single basic block, whereas large kinds are those whose objects occupy more than one basic block.

Scalar kinds contain no references, reference kinds contain only references (i.e. arrays of references) and instance kinds contain reference fields interspersed with scalar fields. Instance kinds need descriptors so that their reference fields can be located by the store (descriptor kind is used for these descriptors). The Napier kind is used for Napier-style objects; these objects have all their reference fields grouped together at the beginning, and the scalar fields following them.

By trying to keep the first version using PJSL simple, we only used one kind. There were some choices here:

- Instance kinds. These require descriptors to be generated when they are saved to the store.
- Napier kinds. These will require that all references be grouped at the beginning of the object, which means that the class loading mechanism in the virtual machine will have to be changed to accomodate this.
- Scalar kinds. It is possible to treat all objects as purely scalar. The only time that the store actually has to locate the references in an object is when store garbage collection is done, so all objects can be treated as scalar objects if no garbage collection is done.

Scalar kinds were chosen for simplicity of implementation. The integration was successful, although some problems were encountered which are presented in the next subsection.

Following the success of getting PJSL working using only scalar kinds, we extended the PJSL support to save different objects as suitable kinds. For class instances to be saved as the Instance kind, each class requires a descriptor to allow the store garbage collector to locate the references within the object. A function to generate descriptors for a certain object kind is registered with `pjsl_setDescriptorGenerator`. This function generates descriptors by using a class definition to mark the locations of references in its instances by setting bits in a bitmap.

We did not use the reference-only types with PLaVa, as the current implementation saves objects with their VM-specific headers attached to them with the result that the saved object is not entirely composed of references. While these headers are not required for "properly" persistent objects, they are needed for objects that are swapped out because these objects will need the run-time information stored in the header when they are swapped back in. The header could be saved as a separate object with a reference to it but we felt that this would overly complicate the process and it would also result in twice the number of objects being written to the store.

6.10.2 Problems Encountered

When using PJSL, we find ourselves facing a potential problem. This problem is caused by the fact that when using PJSL, objects must have `firstWrite` called on them in the order in which they were allocated. For example, if objects A, B and C must all be allocated and written then either A, B and C must be allocated and then written in that order as well, or A must be allocated and then written, B must be allocated and then written and finally C must be allocated and written. This causes a problem for us because of the way we handle object swapping and stabilisation.

Stabilisation is the lesser of the two problems, so we'll address that first. As has been described, when objects are promoted we scan the whole heap and write the suitable objects one by one, allocating PIDs for any of their children that do not yet have them. We do not actually write these children until they are encountered in our heap traversal. This means that allocations and writes are interspersed during the stabilisation process.

To overcome this, we must do all our allocations in one pass and then do all our writes (in the same order) in a second pass. When an object is allocated, an entry for it is pushed onto the *allocation queue*. During our heap traversal, only allocations are done and queue entries are made. When this traversal has finished, there will be an entry in the queue for every object that must be written, containing all the information that is needed to write it. These entries are dequeued and their corresponding objects are written, with the result that the objects are all written in the order in which they were allocated.

Calls to `update` can be interspersed with calls to `firstWrite` although this is "ultimately not advisable". Doing `update` and `firstWrite` calls together did not cause problems when running, but provision was made to avoid this case. When an `update` needs to be done, an entry is pushed onto the queue in the same way that a request for a `firstWrite` is pushed but is distinguishable from them. If an `update` request is encountered before all the `firstWrite` requests have been handled then it is pushed onto the end of the queue. When all the `firstWrite` requests have been completed then the `update` requests are processed as they are the only entries left on the queue.

The problem with swapping is somewhat more complicated. When an object is swapped out, PIDs are allocated for those of its children that need them because they need to be saved as references. These children are not written unless they themselves are swapped out, so allocating PIDs for them will upset the write ordering. Actually writing these objects at this point would seem to overcome this problem, but the problem is recursive in that PIDs will need to be allocated for the child object so that it can be written.

To overcome this, the child object must be written with its references still in memory format and flagged as dirty. This will satisfy the write ordering (at the price of extra writes) and when the object needs to be written during a stabilise, it will be written properly because it is dirty.

6.11 Catering for Alternative Mechanisms in PLaVa$_2$

The new version of PLaVa offers extra opportunities where configuration options can be provided.

6.11.1 Eviction Alternatives

Currently, garbage collection is invoked when an allocation fails. Since garbage collection stops the execution of the JVM, it can be advantageous to limit the number of times it runs by freeing up "sufficient" heap space.

A useful configuration parameter is the percentage of the heap that is required to be freed during a garbage collection/eviction operation. As discussed, there are both advantages and disadvantages in raising the number of objects that are evicted.

In order to save TPOT entries and store space, transient objects can be saved in memory format when they are swapped out so that PIDs do not have to be allocated for all their children. This is not advantageous with objects that are likely to become persistent because in this case the memory-format objects will have to be swapped in during a stabilise so that they can be rewritten in store format. Heap garbage collection can also be performed at stabilisation time.

Stack-directed marking can be used to “de-select” useful objects from categories that have been selected for eviction. There are two parameters that can be changed in this process - the number of frames from the top of the stack to use and the degree of indirection to follow from the stack.

6.11.2 Swizzling Alternatives

The different swizzling schemes described in chapter 5 remain as configuration parameters in PLaVa$_2$; although the effects are different as discussed in the next chapter.

An additional swizzling scheme is available in PLaVa$_2$. Since temporary objects are being swapped to the store and hence the virtual machine is checking handles to see if they contain PIDs, handles can be used as proxy objects [44] for all of an object’s reference fields. This corresponds to eager, indirect swizzling.

To implement eager, indirect swizzling all of an object’s reference fields must be swizzled as soon as the object is faulted in. If the object referred to by a reference field is already resident, then its existing handle is used. To determine whether or not it’s already resident both the RPOT and the TPOT are queried using its PID. If it has an RPOT entry then the LA field of its RPOT entry is used as its handle if it does not have a separate handle. If the object is not resident then either an RPOT entry or a handle is allocated for it depending on whether it’s a persistent object or a transient object that’s been swapped out, respectively. The content of the handle is set to the referenced object’s PID. During execution, bytecodes that use references will not have to check whether or not a field has been swizzled because they will all already have been swizzled.

This scheme is not suitable for large arrays of references since many (possibly unused) handles or RPOT entries would be allocated. The approach taken in PJama [4] is to swizzle such arrays lazily and other objects eagerly. It is easy to implement array element swizzling differently to object field swizzling because objects and arrays are accessed by different bytecode instructions.

6.12 Summary

This chapter presented the second version of PLaVa, which introduced indirect addressing and enabled objects to be swapped to the store as well as allowing the heap to be compacted during garbage collection.

When the heap is full, objects are marked as reachable from the machine, from the RPOT and from the TPOT. Using these markings, all the objects in the heap are categorised according to their eviction overhead and categories with the least eviction overhead are evicted until enough memory has been reclaimed. Stack-directed marking can be used to determine a set of objects that should remain in memory even if they are in a category that is to be evicted.

Results from evaluating different eviction options are given in the following chapter.

Chapter 7

Evaluation

7.1 Introduction

This chapter presents a selection of PLaVa test cases and compares results gained from running with different machine configurations. Most of the tests are concerned with $PLaVa_2$ as it offers many more options than $PLaVa_1$.

The first test application places a map of the London Underground on the store. The map is stored as a graph with nodes representing stations and edges representing connecting lines. The stations are also arranged in a binary search tree so that a particular station can easily be found by specifying its name. Tests include traversals and searches. The size of the database is small, just under 300Kb.

The second test application is a Java implementation of the Generalised Search Tree [63] (GiST), including B-tree and R*-tree implementations. Tests run include insertions, searches for specific elements, searches for ranges of elements and deletions. The database size in these tests depends on the size of the tree used; the 10,000-element B-tree is 1.3Mb.

In order to use a widely accepted benchmark on our system, we did some testing with a Java port of the OO7 benchmark suite [17]. The "medium" OO7 database configuration is the largest data set yet used with PLaVa, at just under 60MB.

Most of the tests presented in this chapter are chosen to highlight the advantages

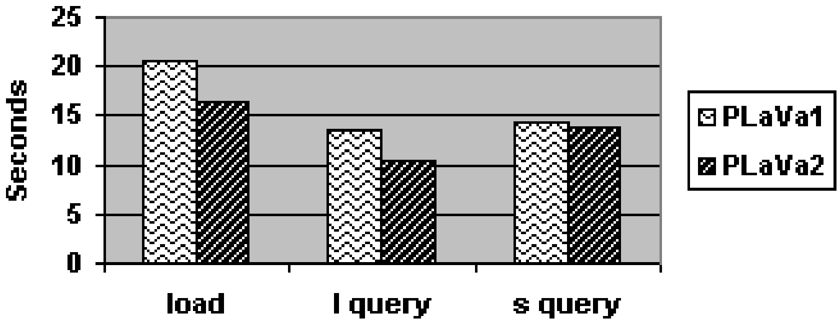

Figure 15: Comparing direct ($PLaVa_1$) and indirect ($PLaVa_2$) addressing.

or disadvantages of different configurations. Unless otherwise stated, swizzling is used.

7.2 PLaVa Test Cases

These tests were run on a Pentium 75 in Windows 95 DOS mode with 32 megabytes of memory (the tests are all constrained to use substantially less memory than this).

7.2.1 Adding Handles

The first graph (figure 15) compares execution times of a query run with the first (direct addressing) and the second (handle addressing) versions of PLaVa. No reachable objects were swapped out, as the direct addressing version does not support this. The second version proves to be faster, the compacting garbage collector resulting in more efficient memory management. The first columns of the graph show times taken to load data from a file and create objects. The second columns ("l query") show the times taken to run the query after loading from the file and the third columns ("s query") show the times taken to run the query off the store (cold query).

The remainder of this section presents tests run on $PLaVa_2$.

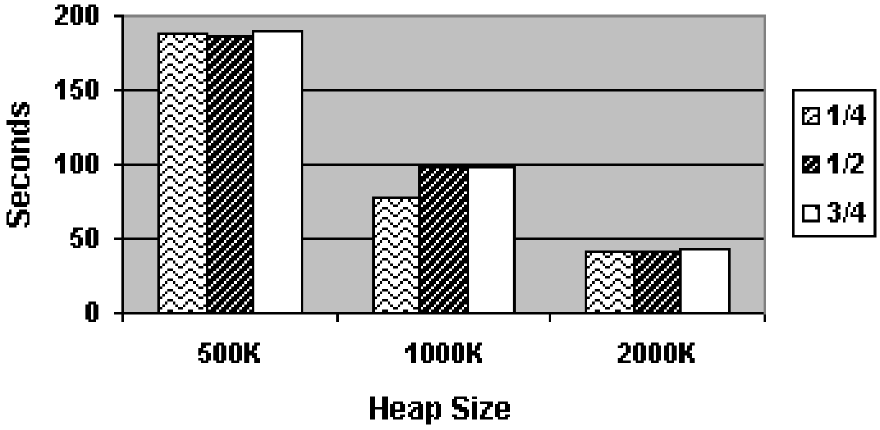

Figure 16: The effects of differing heap sizes and eviction factors on a large search (1000 items in a 10000 item B Tree)

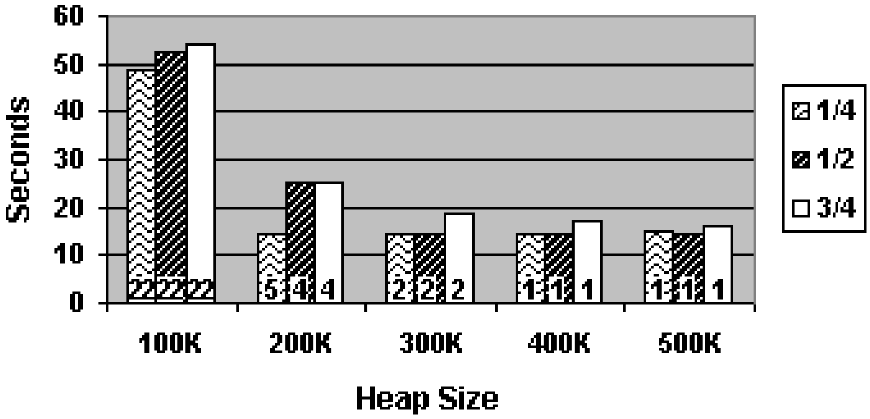

Figure 17: The effects of differing heap sizes and eviction factors on a smaller search (connectivity of stations in the London Underground). The numbers on the bars indicate the number of garbage collection operations done.

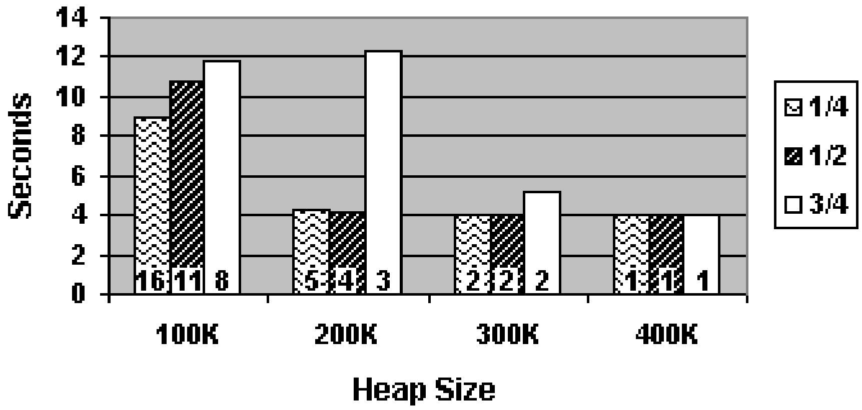

Figure 18: The effects of differing heap sizes and eviction factors on an update (changing names of stations in the London Underground). The numbers on the bars indicate the number of garbage collection operations done.

7.2.2 Heap Size

As would be expected, increasing the heap size allows more objects to be resident at once and, consequently applications execute more quickly. This is evident in the case of a search (figures 16 and 17) and an update (figure 18). Once the heap exceeds a certain size, most of the important (frequently used) objects can fit in memory and execution times become similar.

As can be seen from the number of times that garbage collection is done, it is not the number of garbage collections that is done but the *type* of objects that are evicted that is important in affecting running time. The eviction factor plays a large role in this.

7.2.3 Eviction Factors

If the eviction factor is high then a large percentage of the heap will be freed for use. The consequence of this is that fewer garbage collections are done, but objects that were swapped out and are reused may need to be swapped back in. Tests show that a small eviction factor (1/4 or 1/2) is generally better than a high one (3/4 and sometimes also 1/2) because of this.

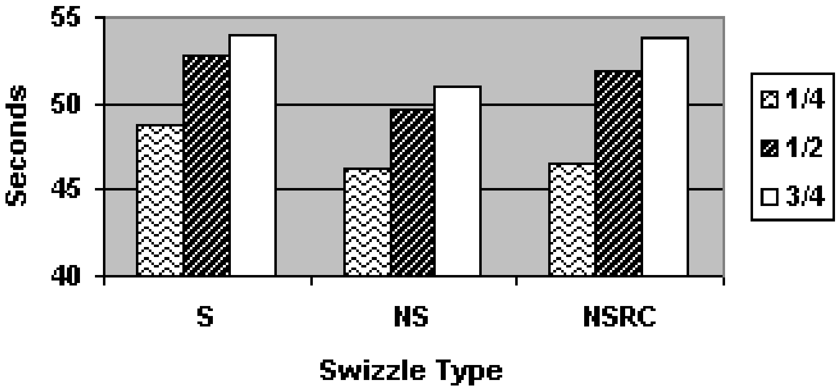

Figure 19: The effect of different swizzling strategies in a small (100K) heap. (S is swizzling, NS is non-swizzling, NSRC is non-swizzling with residency checks during marking.)

The smaller eviction factor allows frequently-used nodes of trees and graphs to stay resident, resulting in better performance. In some cases, the decrease in efficiency from 1/4 to 1/2 or from 1/2 to 3/4 eviction factor is quite marked. To decrease this sudden jump, fine-grained selection should be used to select only the least important objects from the expensive categories to evict.

The spike in figure 18 at 3/4 eviction factor with a 200K heap is evidence that the high eviction factor is causing too many evictions of dirty objects. Many cheap evictions are better than a few expensive evictions.

7.2.4 Swizzling or No Swizzling

In cases where a lot of swapping is done (i.e., low heap size and/or high eviction factor), no swizzling is more efficient than swizzling. This is because objects that are swapped out are in store format (with their references as PIDs) and the extra work done in order to swizzle the references does not pay off. In fact, extra work has to be done in order to deswizzle the references when the object is written out. The loss in efficiency is not substantial, but is highlighted in figure 19 which was done with a 100K heap.

Figure 20 demonstrates the opposite effect when a large heap (500K) is used.

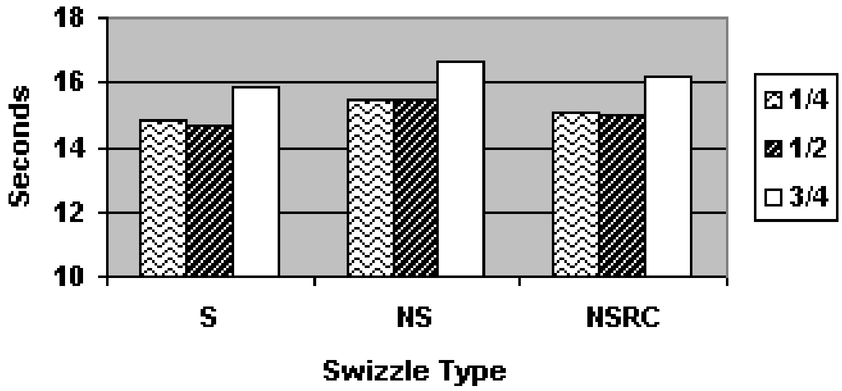

Figure 20: The effects of different swizzling strategies in a large (500K) heap. (S is swizzling, NS is non-swizzling, NSRC is non-swizzling with residency checks during marking.)

Swizzling is more effective here because the work done swizzling the references is less likely to be undone.

Residency checks can be done on PIDs during marking so that more objects can be taken as reachable (i.e. including those resident but only referenced by PIDs). Using this with no swizzling gives a good compromise between the swizzling and non-swizzling versions in both figures 19 and 20.

7.2.5 The Effect of Dirty Objects and Swapping

Figure 21 compares the execution of two very similar programs. The search program searches for stations in the Underground graph and appends some characters to their names but does not record the changes to the station objects. The update program does the same processing and then records the new name in the station object. The effect of the eviction factor on the execution times when dirty objects are present is obvious here, as these dirty objects must be (expensively) written in order to free up more heap space.

As can also be seen from figure 21, it is not the *number* of garbage collections that slows down execution, it is the *type* of garbage collections that is done. Evicting unreachable, clean objects has a negligible effect on running speed, as can be seen

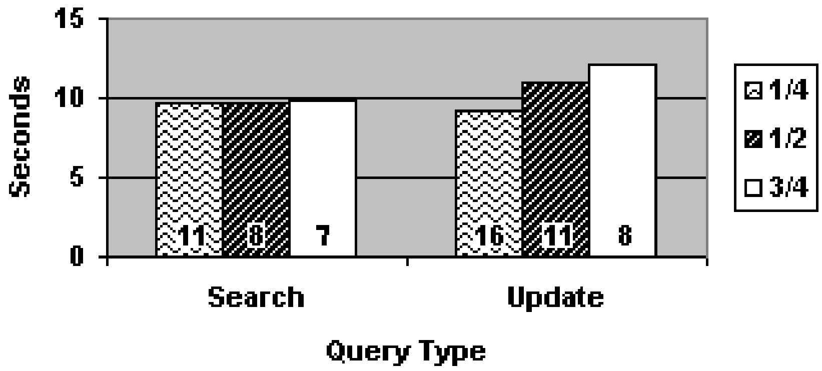

Figure 21: The effects of dirty objects when swapping. The numbers on the bars indicate the number of garbage collection operations done.

from the garbage collection counts for the search query on the left.

7.2.6 Effects of Stack-Directed Marking

To investigate the effects of stack-directed marking, a query was run on the London Underground graph using a 200K heap and a 3/4 eviction factor. This meant a lot of swapping was done and the objects retained would have a larger impact. The query finds all stations reachable from specified stations. The times are arranged with the number of stack frames used listed downwards and the depth of references followed from the stack listed across. For reference, the time taken for the query with no stack marking is 35.9.

	depth				
↓ frames	1	2	3	4	5
1	35.6	35.4	33.6	33.6	33.7
3	35.4	35.4	25.7	25.4	25.2
5	35.9	35.7	25.8	25.6	25.4

The figures indicate that once we mark 3 objects away from the stack then the running times decrease steeply. This number will vary from application to application; in this case `Station` objects contain `String` objects for their names. These `String` objects hold their character data in an array and it is this character data (3 objects away from the stack) that is important when searching for a station.

7.2.7 Stabilising

Since dirty objects can be written during garbage collection (thus making them clean), they do not have to be rewritten during stabilisation unless they have become dirty again. This means that stabilisation time is heavily dependent on how many dirty objects were written during garbage collection. Since the eviction factor controls the number of objects written during garbage collection, this can also affect the final stabilise figure.

The following figures illustrate this phenomenon. The “time” column refers to the total execution time, including garbage collection and stabilisation times. The “gc” shows the time taken to do garbage collection and the “stab” column the time taken for the stabilise done at the end of the program. The program that was timed was the creation of the graph of the London Underground which involves adding nodes to a search tree as well as connecting nodes with lines.

heap	EF	time	gc	stab
100	1/4	97.9	27.7	0.1
”	1/2	98.0	27.4	0.1
”	3/4	97.6	27.4	0.1
200	1/4	32.2	7.2	0.6
”	1/2	42.6	10.3	0.3
”	3/4	42.5	10.5	0.3
300	1/4	26.4	4.2	1.4
”	1/2	29.7	6.0	0.7
”	3/4	35.0	8.1	0.1
400	1/4	23.5	0.2	5.3
”	1/2	26.1	4.0	1.5
”	3/4	29.3	5.3	1.2

As can be seen, when a lot of time is spent during garbage collection (i.e. many objects are written to the store), the final stabilise figure is smaller because there are fewer objects remaining to be written. In the case of the 100K heap, the garbage collection figures are very high because many garbage collection cycles were needed and these required writing some objects many times as they were changed often. The latter reason also affects the larger heaps, e.g. in the 400K heap with 3/4 eviction factor, 5.3 seconds was spent doing garbage collection and there was still another 1.2 seconds spent on the final stabilise.

7.3 OO7 Testing

7.3.1 Background to OO7

The OO7 benchmark suite [17] was designed to test several different aspects of OODBMS systems. Particular aspects that were targetted include the efficiency of different sorts of pointer traversals and different kinds of updates. Also targetted was the efficiency of the query processor but this has little bearing on our system since we're using a single persistent language rather than a number of OODBMS systems.

OO7 was designed to provide a more useful benchmark than the OO1 Benchmark [35] (also known as the Sun Benchmark) and the Hypermodel Benchmark [64] from Tektronix. It was felt that these benchmarks did not test a wide enough range of OODB features such as complex objects and different sorts of traversals. The benchmark is designed to be representative of a generalised CAD/CAM/CASE system.

The benchmark database is made up of a number of *composite parts* which each have attributes `id`, `buildDate` and `type`. Each of these composite parts contructed from of a number of *atomic parts* and also has associated with it a *document*. The document has attributes `id`, `title` and `text` (which contains its actual text). Atomic parts have attributes `id`, `buildDate`, `x`, `y`, `docId` and `type`. They also have bi-directional associations with a (configurable) number of other atomic parts. These associations are modelled with connection objects which have attributes `length` and `type`. The number of these associations between atomic parts is known as the "fanout" of the database.

For the purposes of producing a database for benchmarking, an assembly hierarchy

Parameter	**Small**	**Medium**	**Large**
Atomic parts per complex part	20	200	200
Connections between atomic parts	3/6/9	3/6/9	3/6/9
Document size (bytes)	2K	20K	20K
Manual size (bytes)	100K	1M	1M
Complex parts per module	500	500	500
Assemblies per assembly	3	3	3
Levels of assembly	7	7	7
Complex parts per assembly	3	3	3
Modules	1	1	10

Figure 22: Standard configurations for OO7 benchmark taken from [17]. Each of the databases is available in three different versions, with a "fanout" (number of connections between atomic parts) of 3, 6 or 9.

is created. At the top of the assembly hierarchy is a *complex assembly* object. Complex assembly objects contain either other complex assembly objects or *base assembly* objects. Base assembly objects are at the lowest level of the assembly hierarchy and each contain a number of composite parts. The number of levels in this hierarchy is configurable. Each of these hierarchies is called a *module*.

Most of the sizes and quantities relevant to these objects can be set by configuration parameters. Standard parameters for different database sizes are provided as depicted in figure 22.

The systems tested in [17] are Exodus, Objectivity and Ontos.

7.3.2 Running OO7

I used a PJama implementation of the OO7 benchmark converted from the original E implementation [65] by Laurent Daynès. In order to get it running, some minor changes needed to be made to the interpreter and the benchmark suite. The benchmark implementation uses the JDK 1.1 libraries, so the libraries that PLaVa uses were upgraded because it had been using libraries from JDK 1.0 up to that point. Since the bytecode definition between the versions is the same the new libraries were easily incorporated but required that some new native methods be written.

The JDK 1.1 class libraries include reflection classes that can be used to describe classes, methods and fields. The OO7 implementation by Daynès made use of reflection which PLaVa doesn't yet support, so all reflection calls were removed. This shouldn't impact on performance as reflection was chiefly used to pass methods that needed to be run in transactions and our system doesn't use transactions. Direct calls were made in these cases instead of using reflection.

To use the benchmark with a particular system, calls need to be implemented to access the system's store. We implemented calls for our store in terms of the provided interface. These calls include standard store functionality:

- Setting a persistent root
- Loading a persistent root
- Starting a transaction
- Ending a transaction
- Running a specified method in a transaction
- Opening an observer tool

For our purposes, only the calls dealing with persistent roots are important. As our system does not support transactions, the calls for handling transactions are just implemented as stubs.

The benchmark implementation used does not implement all the traversals and queries of the OO7 benchmark suite. It implements traversals 1, 2a, 2b and 2c, all of which ran on our system. These particular traversals do the following:

- *traversal 1* Traverse the assembly hierarchy and when each composite part is found, perform a depth-first search on its atomic parts.
- *traversal 2a* This traverses the hierarchy in the same way that traversal 1 does, but it updates one atomic part per composite part.

- *traversal 2b* Same except update every atomic part.
- *traversal 2c* Same except update every atomic part 4 times.

Running the OO7 benchmark gives us confidence in the accuracy and reliability of the PLaVa system. Unfortunately, the benchmark makes heavy use of interface methods and invoking these is not as efficient as invoking method of other types.

7.3.3 Measurements

The OO7 benchmark suite was run on a machine with a 233MHz Pentium II processor in Windows95 DOS mode. In [17] measurements are presented for the three systems tested in the study. I'll review some of these measurements so that PLaVa's can be compared to them. Since the platforms on which the systems are running are very different, I've compared trends and relative values rather than actual figures.

Traversal 1

Traversal 1 is designed mainly to test pointer traversal speed. Since we've varied the heap size when running the test, the execution time relies heavily on the amount of heap available. All were run with 1/4 eviction factor as it is most efficient in this case. These runs were done with the fanout 3 database configuration.

heap	cold	warm
1M	88.8	87.8
2M	81.5	79.7
4M	61.4	58.4
8M	36.5	16.4
16M	33.5	15.4

As the figures indicate, by the time we reach a 16M heap, almost all the required objects fit into the heap. With the smaller heap sizes the number of objects that can fit in the heap at one time is a small fraction of the total data set, so the cold and warm times are similar. This latter effect is similar to that found by the authors of

[17] when running traversals on the medium database on all three systems. PLaVa's behaviour when running traversal 1 on the medium database was comparable, with a cold time of 1065s and a warm time of 1002s.

The contrast between the cold and warm traversals reported in [17] was more marked in the cases of Exodus and Ontos but to PLaVa in the case of Objectivity (8M and 16M PLaVa runs taken into account here as the "warmth" of the cache with less memory has little effect).

We ran the traversal with the fanout 6 and fanout 9 databases as well, with heap sizes 8M and 16M which are interesting because of the contrast between cold and warm traversals.

fanout	cold 8M	warm 8M	cold 16M	warm 16M
3	34.3	16.2	33.0	15.4
6	67.9	51.6	52.0	24.6
9	105.2	104.4	70.6	33.8

The execution times rise very steeply as the fanout increases with an 8M heap. This steep rise can in part be attributed to the increase in database size because many more connection objects are required to model the extra associations. In the case of the 16M heap, the times rise more gracefully as more objects can be resident at one time and the increased number of objects has less of an impact.

Like PLaVa, Exodus and Objectivity showed a constant increase in running time as the fanout increased [17] although Exodus's increase was by a smaller amount than PLaVa's. Ontos's time increase was not constant [17] but here the comparative difference between fanout 3 and fanout 9 is similar to PLaVa's.

Changing the degree of stack-directed marking does not have much effect on OO7 runs. This is probably due to the fact that we're doing a traversal over the whole data set, so there are few objects that are beneficial to keep between garbage collection cycles.

Traversal 1 with Eager Swizzling

Section 7.2.4 compared swizzling and no-swizzling performance. Tests were done comparing eager swizzling with swizzling-on-use running traversal 1 with the small database (fanout 3). The eviction factor was 1/4 to make the times comparable with the other traversal 1 runs.

heap	cold	warm	cold eager	warm eager
2M	81.5	79.7	81.8	80.6
4M	61.4	58.4	62.0	58.8
8M	36.5	16.4	34.5	15.6

These figures indicate that eager swizzling is less efficient than swizzling-on-use when a lot of swapping is done (i.e. with a small heap) and the situation is reversed when less swapping is done. We expected this after observing the comparative efficiencies of swizzling-on-use and no-swizzling in different sized heaps (section 7.2.4).

Looking at the 8M case, eager swizzling gives a greater benefit on cold data because nothing has been swizzled yet, which means that the swizzling-on-use version has to swizzle each reference as it uses them. The eager version swizzles all references in an object when it is faulted in; in this case the overhead of swizzling some possibly unused references is minimal compared to the time saved by doing them all at once. In the warm case, all the references that are to be used have already been swizzled in both versions. The efficiency gained by the eager swizzling here is that no checks have to be done in order to determine whether or not references have been swizzled.

The eager swizzling version with the 8M heap approaches the swizzling-on-use version with the 16M heap.

Traversal 2

Traversal 2 has three variations, each designed to test a particular aspect of updating.

The results obtained with the systems presented in [17] vary somewhat in how they are affected by the different updates. Exodus (which logs changed objects) increases in running time slightly from 2a to 2b and again to 2c for the small database. Objectivity and Ontos (which log updates to pages) were both less efficient than Exodus overall and both ran 2b and 2c more quickly than 2a. Run on the medium database, all systems ran 2a most quickly, followed by 2c and then 2b.

PLaVa's performance when running traversals 2a, 2b and 2c on the small database (with fanout 3, 6 and 9) with an 8M heap and a 16M heap is presented in the following table.

fanout	8M 2a	8M 2b	8M 2c	16M 2a	16M 2b	16M 2c
3	34.9	34.9	36.1	33.6	33.8	35.0
6	66.6	65.8	67.6	52.3	52.4	54.2
9	104.1	104.0	105.7	70.5	71.1	73.4

Because PLaVa does not perform update logging other than flagging an object as dirty, updating itself does not have much impact on the running times compared to traversal 1. The writes are all done during the stabilise when the traversal has finished. If dirty objects are swapped out during the traversal then these are written at that point and thus do not have to be written during the stabilise. Of course, when dirty objects need to be swapped out then the execution will slow down because of the extra work involved.

7.4 System Comparisons

The small database OO7 traversal 1 was run using three different systems to compare them. The first was PLaVa with the simple store, the second was PLaVa with PJSL and the third was PJama (version 0.5.7.10). These tests were done on a 170MHz Sun SPARCstation 5 running SunOS 5.5.

system	cold	warm
PLaVa with simple store	66	21
PLaVa with PJSL	56	47
PJama	17	15

There is no apparent reason for the discrepancy between the warm times when running PLaVa with the simple store and with PJSL. It is possible that this is caused by some inefficiencies internal to the virtual machine brought about by the different PID structure. Despite the higher warm time, the cold time is lower than when using the simple store, indicating that PJSL is more efficient. The small difference between warm and cold times running with PJama indicates a much more efficient faulting mechanism than PLaVa's. This is mainly due to PJama's faulting in pages using a two-level cache, whereas PLaVa fetches an object at a time when needed. Another factor is that PJama makes full use of available memory, and can probably cache a large percentage of the relatively small store.

7.5 Summary

The effect of changing the heap size is much as would be expected: the larger the heap, the shorter the running time. Once the heap reaches a large enough size for most of the required objects to fit, further increases do not reduce the running time.

The eviction factor has a marked effect on running time. A smaller eviction factor results in more garbage collection cycles, but these cycles are cheap if they do not need to evict dirty objects. A larger eviction factor usually requires swapping out dirty objects, resulting in more time spent swapping.

When minimal swapping is done, swizzling decreases the running time as would be expected. However, when a lot of swapping is done, the swizzled references have to be deswizzled often, resulting in increased running time. In the former case, eager swizzling is more efficient than swizzling each reference on use but also suffers when lots of swapping is required.

Stack-directed marking selects useful objects not to evict. It is most beneficial when a lot of swapping is done and there are a number of important objects that are used often.

The time taken to do a stabilise depends greatly on how many dirty objects remain in the heap, as some might have been swapped out and do not have to be rewritten. In scenarios where most of the writing is done during a final stabilise, the total running time is lower.

The time taken for PLaVa to perform operations on cached data is about 30%

more than is taken by PJama. Performance on data that needs to be faulted in from the store is not as good, but is better with an efficient store such as PJSL.

Chapter 8

Conclusion

8.1 Summary

This thesis has described a portable, persistent Java virtual machine designed particularly for platforms with very limited memory. Two versions of the machine were presented. PLaVa$_1$ was developed from an existing lightweight JVM by adding persistence through minimal perturbation to the original architecture. This machine has direct addressing and lazy direct swizzling, and promotes objects if necessary when heap space must be reclaimed.

PLaVa$_2$ focusses on memory space reclamation; indirect addressing and garbage compaction are introduced and, when necessary reachable, possibly transient, objects are evicted to the store. The victim selection mechanism is a modification of the standard non-persistent garbage collection algorithm that takes into account the overhead concerned with evicting different categories of objects. Both versions of the machine are built to support alternative persistence mechanisms, so that the performance of different configurations can be compared.

The PLaVa system has run a few purpose-made test applications as well as existing applications which include Generalised Search Tree [63] queries and a number of traversals from the OO7 benchmark suite [17]. Simple applications have been run with a heap as small as 100K. We have demonstrated the adaptability of PLaVa to existing stores by interfacing with the PJSL store that has been developed for use with PJama.

Various swizzling and memory management configurations have been evaluated and the results reported. Although programs have been run with very small heaps, increasing the heap size can dramatically decrease the program's running time. Using swizzling can decrease running time in situations where objects are not evicted too often. Stack-directed marking can be used to retain useful objects in memory.

8.2 Future Work

Future work will include the use of a remote store, so that the PLaVa system can run on a small machine that has no local disk and obtains its data via a network.

Further testing of different machine configurations is also possible. For example, programs need to be analysed [58, 59, 60] to evaluate our strategy of faulting objects in when references are pushed onto the operand stack. Different eviction schemes can be incorporated into the garbage collection strategy for dealing with reachable objects, such as standard cache replacement schemes. Either in conjunction with this or separately, a method needs to be designed to calculate the best eviction factor to use at a certain point in the program. As we have seen from results, this is likely to be the smallest eviction factor that will give us enough free memory to continue execution.

PLaVa can be used with other available stores such as PSI [5] and Shore [66]. Adapting PLaVa to use these stores might be more complex than adapting it to use PJSL because the simple store that was first used with PLaVa has a similar interface to PJSL. Ideally we need to allow any PLaVa program to execute against any of these stores.

PLaVa can be adapted either to interact with or act similarly to Sun's EmbeddedJava [3] system. EmbeddedJava allows the user to use tools to select only the fields and methods that are required to run a particular application. This saves on memory and would be a big advantage in a restricted environment, but may complicate persistence because of the way the class files are processed.

8.3 Related work

The object faulting architecture discussed in this thesis is well known; early examples are PS-Algol [11, 39] and LOOM [12]. PLaVa's implementation probably has more in common with these early systems than it does with other persistent Java implementations as it attempts to use minimal memory.

The language design is similar to that of PJama [4, 20] in that named roots are added to the store and persistence by reachability is used by modifying the virtual machine. In contrast, the JavaSPIN approach [7, 24] makes no modifications to the virtual machine but alters the compiler to introduce new methods to the `Object` class to support persistence.

A study of various swizzling strategies in the context of the Mneme store [67] is given in [32, 44] and in the context of the E programming language [34] in [33]. PLaVa uses a variety of swizzling strategies and their efficiency is compared in different situations.

References to seminal works in the field of persistent languages can be found in [9].

8.4 Closing Comments

Before I started this thesis, I had used persistent languages but had no knowledge of how they were implemented. While developing PLaVa I gained experience and knowledge of a number of topics due to the wide range of research areas available under the umbrella of persistence. These include compiler design, interpreter design, memory and cache management, database management and transaction control.

A lot of work has been done on persistent languages and, considering their immense usefulness, it is surprising that they are not more widely adopted. With the advent of Java, orthogonal persistence can gain more recognition through the commercial popularity that Java currently enjoys. PJama has taken full advantage of this by developing in partnership with Sun Microsystems.

GemStone Systems Inc. has released GemStone/J [6] which provides an environment for using shared persistent Java objects in the middle tier of 3-tier applications. Like most other persistent Java implementations, GemStone/J uses a modified virtual machine to implement persistence by reachability. JavaSPIN differs in that it implements persistence through library support only, which allows it to run on any standard JVM.

With these different approaches, there would seem to be advantage in defining a standard for persistent Java in order to unite the efforts of the various groups working in this area. While this would expedite the production of a robust and efficient persistent Java, it must not restrict research to a particular approach and perhaps invalidate any attempts at trying other, possibly better approaches. While most persistent Java implementations (and to some extent Java itself) are still in their infancy, it would be imprudent to define a standard until more experience has been gained. Even though a system such as PJama has reached "adolescence", there are still a few unresolved issues such as persistence of threads, transactions and handling external state [21].

As stated in the introduction, these implementations are developed for large sytems and large data sets. In this thesis attention has been paid to keeping memory usage down and thus PLaVa has very different goals. These objective have been met by building a persistent JVM that has successfully run a variety of non-trivial programs using very limited memory. Furthermore, we have provided a number of configuration parameters to give some control over the virtual machine operation, and we have demonstrated its ability to work not only with our own store, but also with the PJSL store. We conclude that small machines like DSTV decoders are definitely viable platforms for persistent Java.

Bibliography

[1] James Gosling, Bill Joy, and Guy Steele. *The Java^{TM} Language Specification.* Addison Wesley, 1996.

[2] Stephen Tjasink and Sonia Berman. Providing persistence on small machines. In *Eighth International Workshop on Persistent Object Systems*, September 1998.

[3] Sun Microsystems, Inc. *EmbeddedJava.* http://java.sun.com:80/products /embeddedjava/index.html, 1998.

[4] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design Issues for Persistent Java: a Type-Safe, Object-Oriented, Orthogonally Persistent System. In *Seventh International Workshop on Persistent Object Systems*, 1996.

[5] S. M. Blackburn. *Persistent Store Interface: A Foundation for Scalable Persistent System Design.* PhD thesis, Australian National University, Canberra, Australia, July 1997.

[6] Bob Bretl, Allen Otis, Marc San Soucie, Bruce Schuchardt, and R. Venkatesh. Persistent java objects in 3 tier architectures. In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[7] Alan Kaplan, Geir A. Myrestrand, John V. E. Ridgway, and Jack C. Wileden. Our SPIN on Persistent Java^{TM}: The JavaSPIN Approach. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[8] Tony Printezis, Malcolm Atkinson, Laurent Daynès, Susan Spence, and Pete Bailey. The Design of a new Persistent Object Store for PJama. In *Second International Workshop on Persistence and Java^{TM}*, 1997.

[9] Malcolm Atkinson and Ron Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[10] Michael J. Carey and David J. Dewitt. Of Objects and Databases: A Decade of Turmoil. In *22nd Conference on Very Large Databases*, September 1996.

[11] M. Atkinson, K. Chisolm, and P. Cockshott. PS-Algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, 1982.

[12] T. Kaehler and G. Krasner. *LOOM - Large Object-Oriented Memory for Smalltalk-80 systems*, chapter 14, pages 251–270. Addison-Wesley, 1983.

[13] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[14] J. Richardson, M. Carey, and D. Schuh. The Design of the E Programming Language. Technical Report 824, Computer Sciences Dept., University of Wisconsin, February 1990.

[15] G. N. C. Kirby, A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearle, V. S. Dunstan, R. Morrison, and D. S. Munro. *Napier88 Standard Library Reference Manual (Release 2.2.1)*. University of St Andrews, 1996.

[16] A. Albano, G. Ghelli, and R. Orsini. An Introduction to Fibonacci: A Programming Language for Object Databases. Technical Report FIDE/95/120, FIDE_2, 1995.

[17] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, June 1993.

[18] Ken Arnold and James Gosling. *The Java^{TM} Programming Language*. Addison Wesley, 1996.

[19] J. Eliot B. Moss and Antony L. Hosking. Approaches to Adding Persistence to Java. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[20] Mick Jordan. Early Experiences with Persistent Java^{TM}. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[21] Mick Jordan and Malcolm Atkinson. Orthogonal persistence for java - a mid-term report. In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[22] Misha Dmitriev. The first experience of class evolution support in pjama. In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[23] A. Kaplan. *Name Management: Models, Mechanisms and Applications.* PhD thesis, University of Massachusetts, Amherst, MA, May 1996.

[24] John V. E. Ridgway and Jack C. Wileden. Towards class evolution in persistent java. In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[25] Jon Howell. Straightforward java persistence through checkpointing. In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[26] Olivier Gruber. Transparent Access to Legacy Data in Java. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[27] C. Souza dos Santos and E. Theroude. Persistent Java. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[28] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report (revised). Technical Report DEC SRC 52, DEC Systems Research Center / Olivetti Research Center, November 1989.

[29] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1983.

[30] O2Technology. C++ Interface to O_2, December 1994.

[31] Thomas Atwood. Two Approaches to Adding Persistence to C++. In *Fourth International Workshop on Persistent Object Systems*, 1990.

[32] J. E. B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Computers*, 1991.

[33] Seth J. White and David J. Dewitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *18th Conference on Very Large Databases*, 1992.

[34] Joel E. Richardson. Compiled Item Faulting: A New Technique for Managing I/O in a Persistent Language. In *Fourth International Workshop on Persistent Object Systems*, 1990.

[35] R. G. G. Cattell and J. Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[36] J. Eliot B. Moss and Antony L. Hosking. Expressing Object Residency Optimizations Using Pointer Type Annotations. In *Sixth International Workshop on Persistent Object Systems*, 1994.

[37] Antony L. Hosking. Residency Check Elimination for Object-Oriented Persistent Languages. In *Seventh International Workshop on Persistent Object Systems*, 1996.

[38] Q. Cutts, S. Lennon, and A. Hosking. Reconciling buffer management with persistent optimisations. In *Eighth International Workshop on Persistent Object Systems*, September 1998.

[39] A. L. Brown and W. P. Cockshott. The CPOMS Persistent Object Management System. Technical Report PPRR-13-85, Universities of Glasgow and St Andrews, 1985.

[40] Laurent Daynès and Malcolm Atkinson. Main-Memory Management to support Orthogonal Persistence for Java^{TM}, 1997.

[41] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, 1992.

[42] Francis Vaughan and Alan Dearle. Supporting Large Persistent Stores using Conventional Hardware. In *Fifth International Workshop on Persistent Object Systems*, 1992.

[43] P. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *ACM Computer Architecture News*, pages 6–13, June 1991.

[44] Antony L. Hosking and J. Eliot B. Moss. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 288–303, 1993.

[45] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994.

[46] S. M. Blackburn and R. B. Stanton. The Transactional Object Cache as a Basis for Persistent Java System Construction. In *Second International Workshop on Persistence and Java^{TM}*, 1997.

[47] Laurent Daynès. Extensible Transaction Management in PJava. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[48] Alex Garthwaite and Scott Nettles. Transactions for Java. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[49] S. M. Blackburn and J. N. Zigman. Concurrency - the fly in the ointment? In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[50] Florian Xhumari, Cassio Souza dos Santos, and Marcin Skubiszewski. Java Universal Binding: Storing Java Objects in Relational and Object-Oriented Databases. In *Second International Workshop on Persistence and Java^{TM}*, 1997.

[51] Tony Printezis. Disk Garbage Collection Strategies for Persistent Java. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[52] Susan Spence. Distribution Strategies for Persistent Java. In *First International Workshop on Persistence and Java^{TM}*, 1996.

[53] Stephen Tjasink and Roland Paterson-Jones. A Highly Portable Java Virtual Machine. Technical Report CS-97-03-00, Computer Science Department, University of Cape Town, July 1997.

[54] Sun Microsystems, Inc. *JNI — Java Native Interface.* http://www.javasoft.com:80/products/jdk/1.2/docs/guide/jni/index.html, 1997.

[55] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification.* Addison Wesley, 1996.

[56] J. Hardwick. *Optimising Java for Speed.* http://www.cs.cmu.edu/ jch/java/speed.html.

[57] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[58] D. I. K. Sjøberg, Q. Cutts, R. Welland, and M. P. Atkinson. Analysing persistent language applications. In *Sixth International Workshop on Persistent Object Systems*, 1994.

[59] Stein Grimstad, Dag I. K. Sjøberg, Malcolm Atkinson, and Ray Welland. Evaluating usability aspects of pjama based on source code measurements. In *Eighth International Workshop on Persistent Object Systems*, 1998.

[60] K. Brahnmath, N. Nystrom, A. Hosking, and Q. Cutts. Swizzle-barrier optimisation for orthogonal persistence for java. In *Third International Workshop on Persistence and Java^{TM}*, September 1998.

[61] Tony Printezis, Malcolm Atkinson, and Laurent Daynès. The Implementation of PJSL: A Scalable, Flexible, and Extensible Store Layer for PJama, February 1998.

[62] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Addison-Wesley Publishing Company, fourth edition, 1994.

[63] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. Technical Report 1274, University of Wisconsin at Madison, 1995.

[64] T. L. Anderson, A. J. Berre, M. Mallison, H. H. Porter III, and B. Schneider. The hypermodel benchmark. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *EDBT'90: Advances in Database Technology - Proc. of the International Conference on Extending Database Technology*, pages 317–331. Springer, Berlin, Heidelberg, March 1990.

[65] Michael J. Carey and David J. DeWitt and Jeffrey F. Naughton. *E/Exodus Implementation of the OO7 Benchmark.* ftp://ftp.cs.wisc.edu/oo7/implementations/exodus.OO7.tar, 1993.

[66] M. Carey, D. DeWitt, J. Naughton, and M. Solomon. Shoring Up Persistent Applications. In *1994 ACM SIGMOD Conference*, May 1994.

[67] J. Eliot B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139, 1990.