A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform

Malcolm Atkinson and Mick Jordan

A Review of the Rationale and and Architectures of PJama: A Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform

Malcolm Atkinson and Mick Jordan

SMLI TR-2000-90 June 2000

Abstract:

A primary goal of research into orthogonal persistence is to simplify significantly the construction, maintenance and operation of applications in order to save software costs, extend the range of applications and improve users' experiences. To test such claims we need *relevant* experiments. To mount such experiments requires an *industrial-strength* persistent programming platform. The PJama project is an attempt to build such a platform and initiate those experiments. We report our design decisions and their consequences evaluated by four years of experience. We have reached a range of platforms, demonstrated orthogonality and provided durability, schema evolution with instance reformatting, platform migration and recovery. The application programming interface is now close to minimal, while we support open systems through a resumable-programming model. Our architecture is flexible and supports a range of optimisations. Performance measurements and current applications attest to our progress, but it is still possible to identify major research questions, and the experiments to test the utility of orthogonal persistence are still in their early stages.

Sun microsystems

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
malcolm.atkinson@eng.sun.com
mick.jordan@eng.sun.com

A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform

Malcolm Atkinson[1] and Mick Jordan[2]

June 2000

[1]Department of Computing Science, University of Glasgow, Scotland. mpa@dcs.gla.ac.uk
[2]Sun Microsystems Laboratories, M/S MTV29-112, 901 San Antonio Road, Palo Alto, CA 94303-4900, USA. mjj@Eng.Sun.Com

Contents

Chapter 1

Introduction

This introduction describes the demand for better application programming technology before we present the hypothesis that orthogonal persistence *might* be such a technology. The PJama project was established to test that hypothesis. Such tests require a more accessible and more robust implementation of an orthogonally persistent platform than had previously been attempted. This paper reports on our designs, implementations and experiences with this platform and analyses the initial evidence from the experiments.

1.1 Background and Motivation

The demand for more and better application systems continues to grow. Currently, these applications are often called *enterprise applications*.[1] Almost all of these applications require a combination of sophisticated data models, complex logic and persistence. These three requisites are driven by human ability to establish and comprehend complex enterprises, to form subtle mental models and to persevere at tasks, individually or in teams, over long periods. As communications improve and our society's complexity rises these fundamental driving forces increase.

At present, these requirements are supported by a combination of database systems, programming languages and software production tools. Inevitably, the challenge of building adequate enterprise applications remains demanding as ambitions rise at least as fast as these three technologies improve. Hence any radical improvement in the support for enterprise applications will always be of great commercial significance. The steady increase in affordable computational power removes inhibitions on application complexity. The decreasing cost of computational resources relative to software engineering labour costs means that these resources should be deployed to improve software engineering efficiency.

The current interpretation of "enterprise applications" is probably limited by the available deployment platforms. This has two effects. It constrains the operational model to only permit certain structures and operations. For example, the Enterprise JavaBeans^{TM} architecture [200] envisages a three-tier architecture, restricted computation in thin user-interface clients, and a specific ACID transactional model with long-term data separately managed on legacy relational platforms. It also constrains the kind of applications considered to those that can be made to fit into a relatively simple model of single-user interaction, fixed business rule evaluation and OLTP commitment of operational records to legacy repositories.

There has been a great deal of tuning to optimise these three technologies and there is much highly developed experience on how to map enterprise applications to them. In consequence, any attempt to transfer existing applications from them is confronted with a significant transitional cost. Significant take-up of an alternative technology, even if it offers substantial long-term benefits, is only likely when a new activity

[1] In [22] these were termed "Persistent Application Systems" (PAS).

emerges, so that the transition costs are avoided. A similar delay to take up can be observed with most of the advances in programming language design.

Stonebraker has observed [198] that “many applications are on a path of increasing complexity towards sophisticated data models, complex logic and persistence” and that “current solutions tend to address only the simple business applications.” We here use the phrase “enterprise application” to mean any application that will be needed to support an enterprise. For example, this could be support for collaborative design, for planning, for multi-vendor collaboration in engineering projects or multi-institution negotiation support. A typical example in the future will be an application that selects a significant volume of data according to the actions of its users from an even larger repository and then conducts extensive, user-guided analyses, derivations or simulations. As application complexity increases, flaws in the persistence platform become ever more likely to defeat the ingenuity of application developers. Ultimately, the provision of a consistent orthogonally persistent platform will become essential. Our research provides a clear indication of the form it should take. The question we face today is how to determine whether it would be profitable now?

1.2 Goals for this Review

This review has three functions.

- It amounts to “stock taking” after four and a half years of the PJama project, answering the question “where are we now?”
- It reviews the technical and directional decisions made, for three reasons.
  - To record our choices and rationale as a foundation for comparisons with other work.
  - To inform others, so that they may have a basis for judging our research.
  - To inform ourselves, so that new team members and co-workers are able to orient themselves quickly, and so that we can improve our performance as researchers and engineers.
- It concludes with a set of questions we have uncovered or refined, to guide our deliberations regarding future research directions, and to invite others to participate in the research, as there are far more questions of significant interest than we can find resources to investigate.

1.3 Overview

The paper is in three parts.

❶ In the first part, sections 1 to 4, we set out the context, goals and requirements of the PJama project;

❷ In the second part, sections 5 to 11, we set out the properties of the current release and explain the decisions that led to them; and

❸ In the third part, sections 12 to 16, we assess both the current status and our experiences with the technology and conclude with a set of research questions.

As we attempt to cover virtually every aspect of PJama in this review, it is impossible to present full technical detail. We therefore refer our readers to many other papers and reports. We hope that this “structured bibliography” will help those starting a research or development project in this area.

We are aware that many readers will not wish to persevere through the whole document. We have deliberately repeated cross-references as we introduce the interplay of aspects of our research. We hope that these will allow readers to start where their interests take them, and to expand their reading as they wish. For example, some may find it helpful to scan the summary of our decisions ($\leadsto$15) and then visit our rationale that defends them. Others may wish to start at our research challenges ($\leadsto$16).

The hypothesis that orthogonal persistence will improve enterprise application life-cycle costs or enable a richer set of enterprise applications, is introduced ($\leadsto$2). PJama is introduced as an orthogonally persistent platform ideal for testing this hypothesis ($\leadsto$3). We present a set of requirements and illustrate that these are *all* necessary, but difficult to meet *in combination*. Developments during the PJama project are presented ($\leadsto$4).

PJama implements the specification of “Orthogonal Persistence for the Java$^{\text{TM}}$ platform” (OPJ) [121], described in ($\leadsto$5). The API delivered by OPJ is extremely small and simple and supports *resumable programming*. We discuss the choices that led to the current architecture, e.g., managing objects rather than address spaces or pages, and operating with one virtual machine rather than many ($\leadsto$6). The Sphere storage system enables object management, recovery, and evolution ($\leadsto$7). PJama supports class evolution ($\leadsto$8), an essential adjunct to persistence, that provides the ability to edit the set of persistent classes and transform their populations of persistent instances in a single off-line, atomic and fault-tolerant operation. Applications can be migrated to a new platform ($\leadsto$9).

Transactions enable large systems to be built more easily, but require new semantics and technology in order to execute against a population of objects as an integral part of a language ($\leadsto$10). PJama extends Java$^{\text{TM}}$ Remote Method Invocation (Java RMI) to combine persistence with distribution ($\leadsto$11).

The earlier sections compare aspects of PJama with contemporary research. Section 13 changes focus to compare approaches as a whole, including: the ODMG model and its developments, and automated mappings between Java programming language and relations. The conclusions ($\leadsto$14) summarise the major decisions ($\leadsto$15) and provide a list of open research questions ($\leadsto$16).

Chapter 2

Testing the Orthogonal Persistence Hypothesis

In a massive global industry of great economic and social importance, such as the applications software industry, it is necessary to be sure that the engineering on which it depends is well founded. Those working with orthogonal persistence ally with Stonebraker in believing that the enterprise application industry's present underpinning is showing signs of stress. They choose to approach this challenge by attempting a holistic design of the whole platform that supports enterprise development and deployment [25]. In this section, we restate the claims of orthogonal persistence and propose a test of their validity. We ask whether these claims have already been tested adequately and remain aware that the test's outcomes will vary with time. For example, the capacity of computers and the complexity of applications are both growing. This trend is likely to increase the benefits of providing a new, coherent platform to support enterprise applications. Recent and new tests are therefore better indicators for technology strategists.

2.1 The Orthogonal Persistence Hypothesis

Orthogonal persistence is the provision of equal persistence support for *all* data. That is, the data all have the same rights to longevity and brevity irrespective of their type. Similarly all code looks the same and has exactly the same semantics, irrespective of the longevity of the data on which it is operating. The principles and history of orthogonal persistence research are summarised in [22]. Orthogonal persistence is provided by a platform that supports all of an application's computation. Its effect is to permit application programmers to work without having to consider mappings between persistent and active data representations and without having to explicitly manage data transfers between applications and storage systems. This is referred to as an abstraction over stores [47] and as a single-level store [57] (page 224).

Researchers have long hypothesised that orthogonal persistence will yield a radical improvement in enterprise applications [13, 12, 137, 22, 33]. We will call this the "OP hypothesis". Their arguments are based on the reduction of conceptual complexity that is achieved by minimising the number of implementation technologies that must be mastered, by reducing the need to visualise several different models of an application's data, and by substantially reducing the application code that must be written. Inconsistencies in the behaviour of subsystems, often require much application code to "paper over the cracks". These inconsistencies are often exacerbated when the enterprise application or the supporting equipment is under stress, for example overloaded or subject to partial failure.

In principle, a holistically designed platform, which meets all of the requirements, should not require this discontinuity handling code. This approach posits one coherent computational model that provides the combined functionality of support technologies and an implementation of a platform that delivers that model with sufficient performance. It is also argued that end users benefit because the implemented applications present more consistent behaviour [12].

The benefits of orthogonal persistence are expected to be more significant as application scale and complexity increases. Furthermore, they are expected to be more significant as we examine a longer period in an enterprise application's life-cycle. Persistence is a longevity phenomenon.

Orthogonal persistence researchers do not claim any exclusive ability to improve the software engineering support for enterprise applications. Contemporaneous research yields improvements in type systems (or in data models), in communication support, in language levels, in rapid application generation from high-level descriptions of certain application classes and in build management. Such advances are complementary to orthogonal persistence, which can combine with or facilitate such developments. Programs written to exploit orthogonal persistence will still need to communicate with legacy systems and databases.

2.2 Requirements for a Valid Test of the OP Hypothesis

An adequate test of the OP hypothesis that orthogonal persistence provides significant benefits to enterprise application engineers requires that the following criteria be met.

- ❶ The development team constructing and maintaining the application be independent of the platform developers.[1]
- ❷ The application(s) be typical of real or potential enterprise applications (in scale, complexity and longevity).
- ❸ The trial and observations be conducted over a sufficient period that long-term effects are apparent.

These criteria are all challenging, even though it is a matter of judgment whether they are met. The previous work on persistent programming languages and platforms is extensive. In our judgment, it hasn't met these criteria fully for one of two reasons.

- ❶ The persistence technology was inadequate for that task, e.g., was limited in scale, performance, recovery or evolution mechanisms.
- ❷ The applications or application programming community was inappropriate, e.g., they were closely connected with the technology providers or had insufficient time and resources.

Sometimes both problems occur, as it is impossible to overcome the second difficulty when oppressed by the first. These primary problems often lead to the inability to sustain an experiment of sufficient scale or duration.

2.3 Previous Orthogonal Persistence Experiments

An examination of earlier orthogonal persistence projects is instructive. There are a group of persistent programming languages that have flourished in the academic community but have failed so far to influence enterprise application construction because they weren't based on popular languages.[2] This has two deleterious effects: potential trials have to overcome the shortage of programmers adept in the language and these

[1]We use "platform" as short for "orthogonally persistent platform".

[2]This is not to say that their long-term effects will not be significant; they equipped us for this project.

languages do not have the investment that leads to rich libraries and high-quality implementations. The first example, PS-algol [13] pioneered orthogonal persistence, and was used extensively in a few contexts. For example, it was used for data model [99, 132] and query language experiments [52], and at ICL to implement a set of process modelling tools.[3] The latter should have proved a useful test of the hypothesis, but the experiences were not analysed and published, and are no longer recent.

Napier88 [159] involved an experimental type system, the unfamiliarity of which limited its adoption. Nevertheless, it was probably subjected to the most extensive evaluation of any of the academic research languages [25]. Other persistent languages experiments with advanced type systems include Machiavelli [171, 37], Fibonacci [4], Tycoon [147], PM3 [101], Theta [141], BETA [144, 131], VML [130]. Leontiev [134] has recently made progress with such type systems. These all seek to achieve both persistence and a comprehensive type system covering polymorphism and inheritance. Their novel type systems, and their relatively short supported lives have militated against their use in extensive evaluation experiments.

Another strand of persistent programming language investigation is the provision of a transactional model for PPLs. The two examples we know of are: Venari [94] and TJava [86]. The former is derived from Standard ML, as are many of the languages investigating type systems. The choice of ML for the former and the brevity of investigation for the latter may have inhibited their independent evaluation in applications. Argus [139] was primarily developed to assist programmers in writing distributed systems. However, from the viewpoint of one process that fails and recovers, those that have continued exhibit persistence. Hence Argus explores persistence and provides a transactional model of persistent state change. Similarly, the later work by Liskov on Thor [140] developing a distributed highly reliable object store, further develops transactional persistence in their programming language Theta [141]. The sustained research at MIT on this theme has clearly developed in depth experience with transactional persistent languages, but we are not aware of any report assessing its impact on application construction and maintenance.

The preceding languages, while valuable research vehicles, did not appeal to commercial and industrial application builders because they were unfamiliar. In contrast, persistent languages based on C and C++ have the advantage of familiarity. Examples are: E [181] and O++ [2, 1]. These languages also failed to lead to full evaluations of persistence. We suspect that this is because the weak control of pointers in C meant that they were never able to become sufficiently simple and robust. Persistent extensions to Pascal [49, 48] only became available as Pascal was on the wane. A general purpose approach was the Texas store [191], which uses memory-management techniques to manage swizzling and page faulting, and could therefore bring durable and atomic commitment to legacy C and C++ code. We believe that it did not investigate any of the other persistence requirements, such as evolution.

A number of orthogonally persistent operating systems have also been built. Monads [184] required its own capability hardware, which was only produced in experimental numbers. L3 with its own language was extensively used and good results were reported [137]. However, it remained localised and specialised in its use, possibly because of its idiosyncratic language. The Mungi single-address space operating system [98] is a development of that work, that solves some of the administrative problems by establishing an economic model — resources are retained as long as you keep paying the rent. The Clouds operating system [56] provided persistent objects and persistent threads on a distributed network of standard computers. Similarly, the KeyKos [84] and Eros [190] operating systems offer *transparent persistence* for objects. Grasshopper [71] is a general purpose persistent platform. The challenge of getting applications built on an unfamiliar operating system, inhibited the use of persistent operating systems as vehicle with which to test the OP hypothesis. It seems probable that operating systems supporting persistence will become accepted, but we suspect that most of these systems have insufficient information about an application's data to perform disk-space reclaimation, evolution and migration ($\rightsquigarrow$7, $\rightsquigarrow$8 and $\rightsquigarrow$9). However, once they are available, they will provide the majority of the runtime of a persistent language automatically. Their consistent and universal

[3]The PROCESS WISE toolkit was built and ran on this technology for a number of years.

treatment of all data as persistent should make it relatively simple to implement a consistent and complete orthogonally persistent language. An example of this was reported by Dearle [72].

The work by David Lomet finesses the problem of unfamiliarity by building a substrate for applications on top of a standard operating system [142, 26, 143]. The intention is to allow legacy and new applications to run transactionally and with full recovery, by capturing the state of programs and the interaction events between the programs and their environment. Programs can be run forward from a checkpointed state by replaying input events and verifying that their output events repeat correctly. Though aimed at fault-tolerance and rapid recovery, this serendipitously provides a platform that would make it trivial to make a language orthogonally persistent. The extent to which this has yet demonstrated the efficacy of persistence is unknown. Such a demonstration, for legacy systems, is one of the stated objectives. But this group's approach means that the structure of an application's data and its relationships with code are not known by the persistence platform. This will again limit the support available for disk-space management, evolution and migration.

2.3.1 Summary of Prior Attempts to test the OP Hypothesis

In summary, various difficulties have prevented a full test of the OP hypothesis. Some attempts suffered from the contingent nature of history (for example, crucial members of a team moved or continued funding required a change of direction), many were unable to attract independent users, and partly as a consequence, the resources were often unavailable to build all of the technology that was required to support large and long-lived enterprise applications.

It is unfortunate that insufficient resources are deployed for the long-term evaluation and testing of basic software engineering hypotheses of great significance to the industry [209]. Taking Tichy's advice, we should design, plan and conduct appropriate experiments. As this has not yet been attempted systematically for the orthogonal persistence hypothesis, the protocol for such experiments may need to be established via smaller trials (Challenge 1 on Page 74).

Chapter 3

Orthogonal Persistence Platform Requirements

In this section, the original idea behind the PJama project is introduced, which leads to a set of challenging requirements.

3.1 PJama as a "Test Vehicle" for the OP Hypothesis

The PJama project was born through a desire to test thoroughly the OP hypothesis. There was an opportunity presented by the advent of the Java^{TM} programming language and a potential client team who were attempting to build a typical advanced enterprise application. Even at the start of the PJama project (October 1995), the Java programming language looked as if it would become a popular language for enterprise application implementation. This popularity has the consequent advantage of significant investment in its implementation, in libraries of classes and in tools, and a cohort of experienced programmers. The Java programming language also has a strong type system, which gives sufficient guarantees that a reliable orthogonally persistent platform can be implemented.

Because the Java programming language was new, we hoped that we would be able to capture the interest of applications programming teams, since they would not have previous habits from combining databases or file systems with Java programming language.[1] We therefore hoped to gain the independent evaluations that we needed. This was apparently guaranteed by the Forest project,[2] a project at Sun Labs, that was using persistent object systems to develop a system to support the construction of software by distributed, multi-vendor teams [122, 123, 212, 213]. That sequence of papers illustrates that the PJama project has at least one demonstration of the efficacy of an orthogonally persistent platform.

3.2 Requirements for an Orthogonally Persistent Platform

If the orthogonally persistent platform, PJama, is to fulfill its role as a support for realistic experiments and to capture the trust of application programmers, it has to meet the following requirements.

A **Orthogonality**: All the data (i.e., classes and class instances) that any of the application programmers wish to use must have the full rights to persistence. The only way of ensuring this is to properly implement orthogonality.

[1]This was a misconception on our part. Programmers transferred their habits from other languages.
[2]Later this aspect of Forest was called JP.

B **Persistence Independence**: The language must be completely unchanged (syntax, semantics and core classes) so that imported programs and libraries of classes work correctly, whether they are imported in source form or as bytecodes. This also has the benefit that programmers can move freely between persistent and standard versions of the Java platform. The type system must be just as robust and pointers just as reliable in the persistent context as in a standard context. We interpret this to mean that the orthogonally persistent platform must comply with the Java^{TM} Language Specification (JLS) [91] and should be capable of running the compliance test suites. It should also improve on, or at least maintain, the security provision in standard Java platforms.

C **Durability**: Application programmers must be able to trust the system not to lose their data. Hence recovery mechanisms to handle failures are necessary.

D **Scalability**: People need to be sure that if they commit to using this platform, they won't encounter limits which prevent their applications from running.

E **Schema Evolution**: Once an application is operational and is populated with objects (classes and instances of classes), changes will be required to those classes. As the classes are changed, their instances must be transformed to match. There must be support for any required change. Other conditions must also be met, particularly requirements C, D and I, and the final state must comply with the Java programming language's semantics [91].

F **Platform Migration**: Applications must not be cut off from new underpinning technology (hardware, operating system, or implementations of the Java^{TM} virtual machine (JVM) [138]).

G **Endurance**: The platform must be able to sustain continuous operation as many enterprise applications are in use continuously. For example, a globally distributed team using JP to build an application written in the Java programming language ("Java application") may be working continuously. A target is 7*24 operation.

H **Openness**: Enterprise applications frequently interact with each other, and with external and legacy systems. This introduces a requirement for an open system, whereas many of the systems cited in section 2.3 were closed.

I **Transactional**: Application programmers must be able to achieve transactional operation, or some equivalent, that makes it straightforward to achieve atomic operations, and appropriate isolation from and interaction with other operations.

J **Performance**: The performance must be acceptable. This means that it must not be markedly less than rival strategies for building enterprise applications and, if possible, it should be better. The significance of speed varies, e.g., in some telecommunications billing operations, speed is so critical (100,000 updates per second) that conventional databases cannot be used [55], whereas in others, such as providing interfaces for employees to business processes, the speed and cost *of implementation* are the crucial factors [167]. At present, orthogonally persistent platforms address the latter rather than the former.

3.3 Discussion of Requirements

The set of requirements given in 3.2 is difficult to meet in full. They cover very nearly all of the requirements on a good development programming language and on a good database management system (DBMS). Three omission deserve comment.

❶ *Query Systems* — Our past experience suggests that it is reasonable to expect to build these within the language [99, 132, 52, 21, 24, 124, 162]. We agree that bulk operations and queries are useful. But they are useful over any large, regularly structured data structure, irrespective of its location and longevity.

❷ *Indexes* — Again, we hold that these are best built within the language. They then gain the portability of implementations at that level. But this clearly depends on efficient implementation, and particularly on being able to specify special transactional rules ($\leadsto$10).

❸ *Constraints* — These specify invariants and are clearly useful in achieving correct and reliable systems. Once again, this as relevant with a large program or long-running program's main-memory data structures, as it is for data that has greater longevity. Consequently, we see it as another issue orthogonal to persistence, though we are glad to see relevant work [50] using our platform.

The challenge of meeting all ten of these requirements (A to J above) is that they are sometimes difficult to meet together. For example, allowing schema evolution between *any* two consistent sets of classes with corresponding instance transformations, while guaranteeing durability (C) and achieving endurance (G) is so challenging that it is not achieved by most commercial object-oriented DBMS (OODBMS).

Some requirements simply demand a lot of implementation effort, such as A, F, H and J. Some require semantic innovation to combine, for example openness (H) and orthogonality (A). Some require careful development of algorithms, such as C, D, G and J. Some still present fundamental research challenges, such as the simultaneous satisfaction of B and I.

Chapter 4

The PJama Project's Progress Reviewed

Here, the history of the PJama project is set out and reviewed. This is, in part, to establish a nomenclature for the rest of the report and in part to identify the current status, so that it can be discussed throughout the document. Our work has involved two major components.

- A Persistent Object Store (POS), and
- A modified JVM, called a Persistent JVM (PJVM), and derived from various versions of the Java$^{\mathrm{TM}}$ Development Kit (JDK$^{\mathrm{TM}}$).

Both went through several major redesigns and their relationship changed. We will discuss the extent to which the various prototypes have met our requirements in 4.2; this is summarised in Table 4.1. The investment of effort during the project is analysed in appendix A.

4.1 History of the PJama project

The PJama project[1] began as a collaboration between the Persistence and Distribution Research Group (PADRG) at Department of Computing Science at University of Glasgow[2] and the Forest Research Group (FRG) at Sun Labs,[3] in October 1995. We initially envisaged PADRG building orthogonally persistent platforms (OPPs), and the FRG using them to build JP.

The design and first prototype of PJama (we'll call it PJama$_{0.0}$) was completed by the PADRG by July 1996 [20, 16]. It was based on the JDK1.0 and used the architecture summarised in Figure 4.1.a. Durability and atomicity depended on Recoverable Virtual Memory [188], there was a buffer pool into which pages were brought and object addresses on disk (PIDs) were byte offsets from the start of the file. There was a separate object cache, in which objects appeared to the PJVM as if they were on the standard garbage-collected heap (`gcheap`). This version had a stop-the-world disk garbage collector [173]. It was initially built to run on a Solaris$^{\mathrm{TM}}$ and SPARC$^{\mathrm{TM}}$ architecture combination, but was ported to run on Solaris and WINDOWS/NT Intel X86 platforms.

As PJama$_{0.0}$ had no eviction from the object cache, it quickly stalled with long-running applications, such as a Geographic Information System (GIS) ($\rightsquigarrow$12.1 ❶). To repair this PJama$_{0.1}$ was released in early 1997. This effectively recycled the space occupied by non-mutated objects[4] [62].

[1]Originally called the 'PJava project', but that name subsequently was used to refer to PersonalJava.

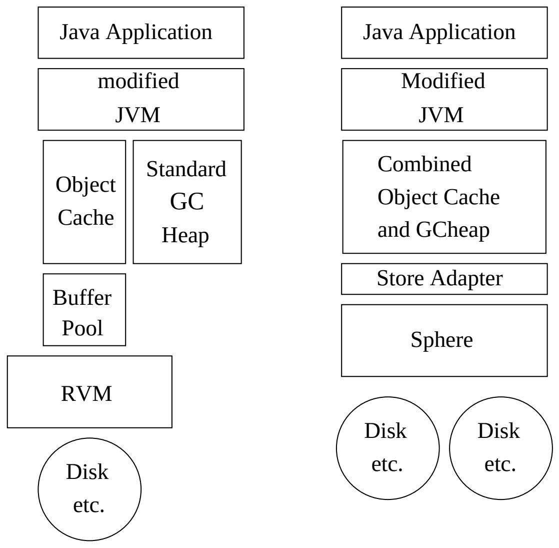
[2]This group had 16 years of experience with implementing orthogonal persistence at that time.

[3]This group had already built early versions of JP using file systems and an OODBMS.

[4]Recoverable Virtual Memory prevented us operating a steal policy on mutated pages, so we could not evict mutated objects as this would have quickly jammed the disk buffer pool with mutated pages.

Minor releases tracked the successive versions of the JDK and added improvements to algorithms and orthogonality. Our next major step was the introduction of a stop-the-world eager evolution system [76, 78] ($\rightsquigarrow$8) in PJama$_{0.2}$ released in August 1998. This version also included our first release of persistence support for Java RMI [196, 194] ($\rightsquigarrow$11). By now, we were able to run a persistent version of the application that demonstrates the ($\rightsquigarrow$12.1 ❷) Swing user-interface components and a substantial GIS system ($\rightsquigarrow$12.1 ❶) . The final versions of PJama$_{0.2}$ correspond with JDK1.1.7 and JDK1.2.

The limits of offset addressing and other constraints on scalability, led to a radically new architecture for PJama$_{1.0}$, which is illustrated in Figure 4.1.b and discussed throughout this paper ($\rightsquigarrow$6 and $\rightsquigarrow$7). The major changes are a new subsystem for durable storage, Sphere [180, 179], and a merge of the `gcheap` and object cache. It was constructed using the Sun Labs Research Virtual Machine (SRVM)[5] [219], which offered significantly improved pointer tracking, better memory management and a JIT compiler [135]. PJama$_{1.0}$ has more extensive durability mechanisms, is more scalable, and has a more comprehensive evolution technology. Inter-platform migration is supported in that release ($\rightsquigarrow$9).



(a) architecture for PJama versions 0.0, 0.1 and 0.3 (b) architecture for PJama version 1.0

Figure 4.1: Persistence Architectures used for Versions of PJama

The original proposals for PJama included a flexible transaction mechanism [20, 64]($\rightsquigarrow$10). The first experimental investigation of this was developed in summer 1998 by running JavaInJava (JIJ) [205] on PJama$_{0.1}$, as PJama$_t$. This gave full orthogonality but paid a high penalty in performance. Work is now underway to incorporate the flexible transaction model into the main-stream persistent platform without undue performance penalties [59]. We anticipate that this will deliver significant benefits to enterprise applications as it will make it tractable for programmers to implement managed interaction between mostly isolated transactions.

A version, PJama$_{pt}$, for the Palm connected organiser was developed during the summer of 1999, based on the Spotless virtual machine [206]. This totally interpreted system also achieves full orthogonality, with persistent threads.

The original modus operandi of PADRG as persistent platform supplier and FRG as persistent platform evaluator quickly evaporated. Several of the team at FRG became implementers of OPPs, and additional

[5]Previously known as the “ExactVM” or EVM. The EVM supported the Java 2 SDK Production Release for Solaris.

platform builders were recruited there. This included members of the PADRG interning with the FRG, and one, Laurent Daynès, a leading engineer, transferring from Glasgow to Sun. Whilst this accelerated the platform development, it eliminated the independence of the trial of orthogonally persistent technology. In truth, the volume of changes in the language implementation, that had to be tracked to meet requirement F would have overwhelmed the PADRG, if the FRG hadn't shouldered that responsibility from 1998 onwards.

4.2 Discussing the Prototypes

We here consider the extent to which we are meeting the requirements set out in ⇝3.2, while in the appendix (⇝A) we consider the effort expended so far seeking to satisfy them. The matrix shown in Table 4.1 summarises the status at April 2000. The values are highly subjective, but 100% is intended to be sufficient to satisfy fully the OP hypothesis testing.[6] There is not intended to be any adjustment for importance, which is discussed in the notes below, with one note per requirement category (row of the table).

Note that none of the existing versions (columns of the table) gets a universally satisfactory score. However, the reasons for falling short of the requirements can vary. Most significant are the cases where we are not yet sure what should be done. Next in importance are the cases where we know what we want to do, but are unsure of a good way of doing it. Minor, but not ignorable, are the cases where we haven't done something yet, because of lack of resources (i.e., it hasn't exceeded the priority of other tasks). The notes below attempt to differentiate these cases. The column labelled "Ref" identifies the principal section of this report where the requirements in a particular row are considered in more detail.

		PJama version						
Requirements		PJama$_{0.0}$	PJama$_{0.1}$	PJama$_{0.2}$	PJama$_t$	PJama$_{pt}$	PJama$_{1.0}$	Ref
A	Orthogonality	75%	80%	85%	100%	100%	90%	5
B	Independence	95%	95%	95%	100%	100%	100%	5
C	Durability	95%	95%	95%	95%	60%	95%	7.3
D	Scalability	60%	60%	60%	0%	90%	90%	7
E	Evolution	0%	0%	75%	0%	0%	90%	8
F	Migration	0%	0%	0%	0%	0%	70%	9
G	Endurance	10%	30%	30%	0%	0%	60%	7
H	Openness	0%	25%	40%	0%	0%	60%	11
I	Transactions	10%	10%	10%	100%	0%	40%	10
J	Performance	80%	80%	80%	5%	70%	95%	12

Table 4.1: Table showing Status of Requirements versus Versions at April 2000

4.2.1 Requirement A — Orthogonality

Orthogonality is a major issue. In all PJama versions, all Java class libraries without native code or references to external state, that is, the majority, persist without difficulty. Where there are even a few classes whose instances are not given the same rights to persistence, applications programmers have a remarkable propensity to trip over them. The most notorious problems arise with GUI classes (mainly in the `java.awt` class library) which prove popular and then entrap programmers into struggling with how to avoid making

[6]The evaluation is based on the assumption that there are no bugs, i.e., deficiencies are due to not having done something yet or not knowing how to do it yet.

them persistent and how to build their own abstraction of the interface's state that they can preserve and reconstitute. This re-introduces the two models and the translation code as a responsibility of the application programmer. *Two models* are precisely what we set out to eliminate! Consequently, much effort has been invested in fully supporting all aspects of the Swing components, `javax.swing`.

The other classes that are likely to prove important for serious enterprise application trials are the `java.jdbc` and `org.omg.corba` class libraries. These problematic classes are associated with the openness requirement (H) and therefore have intrinsically transient state [176]. It is essential that all classes that the implementers of an enterprise application require to persist are properly supported. The obvious culprits (named above) should be dealt with promptly. To avoid tedium overcoming the arbitrary legacy problems, the remaining classes can be dealt with as problems are encountered to achieve a reasonable approximation to orthogonality ($\leadsto$A heading F). In this context, "dealing with" typically means providing code to capture an abstraction of external state or a mapping of some C data structure, on a checkpoint, and code to perform reconstruction on a resume ($\leadsto$5.1).

There are too many classes in the core Java^{TM} class libraries that have C code or initialization dependencies.[7] In many cases this is a legacy of the development of the Java platform or is a logically unnecessary local optimization, better dealt with by improving the JVM implemention. It is straightforward, but time consuming, to remedy these by re-implementing in Java. In the cases where there is intrinsically transient state, the techniques of resumable programming and event handling deal with them satisfactorily ($\leadsto$5.5). The improvements across the first three versions correspond to the introduction of these techniques and their deployment for sockets, `java.rmi` and some of `java.awt`.

The class `java.lang.Thread` has proved a more fundamental difficulty. Its intertwining with the JVM (and even the underlying processors) makes it difficult to achieve persistence, except in the cases where a JVM has been written to model the whole of a thread's state in a form compatible with the Java programming language (PJama$_t$ and PJama$_{pt}$). It might, at first sight, seem unlikely that an enterprise programmer would wish to make a thread persist. However, threads often persist through reachability and their persistence facilitates workflow modelling [148] and many other applications. Ineluctably, the model of continuous execution (precisely the model required by enterprise servers — $\leadsto$5.1), requires that threads and their associated metadata (locks, signals) can persist.

It is not yet obvious how instances of class `Thread` should be handled. An ideal solution would establish a platform-independent model of threads, and arrange that a concrete thread can be halted and then extracted into this model automatically. Similarly, from an instance of this model, a thread can be reconstructed and resumed at the point where it was halted. It is unlikely that this can take place at an arbitrary point, because native machine state is not always accessible, and it is certainly not easily translated into state that can be reconstituted on different platforms. So threads would have to progress to an extractible-point and then halt. Any other model seems impossible to reconcile with evolution (E) and migration (F). Development and validation of a satisfactory, platform-independent resumable-thread model requires research (Challenge 3 on Page 75). For example, migration requires the conversion of the thread state to an abstract model in case the thread arrives on a different architecture and evolution requires enough information to rearrange the stack when a method changes, or to inhibit such a change. Improved operating-system support could enable the state extraction and resurrection.

4.2.2 Requirement B — Persistence Independence

Independence, which is critical for convenient code re-use, is assured if we achieve exact compliance with the (JLS [91]). In the first three versions of PJama the initialization of classes was over-eager and therefore there was not exact compliance [120]. This has been remedied. There are some places where the definition

[7] They assume their initializer is run every time that a JVM execution loads them.

of Java platform is not explicit about the semantics of resumption in the context of a continuous execution model. This requires clarification [120, 121]. Otherwise, we can be reasonably confident that there are no unsolved problems with persistence independence.

4.2.3 Requirement C — Durability

Any persistent platform must provide application developers with confidence in its reliability. That is, they must be assured that their software failures, platform-software failures, and hardware failures will not lead to loss of data. In versions PJama$_{0.0}$ to PJama$_t$ the underlying Recoverable Virtual Memory (RVM) technology provides recovery from the first two forms of failure. In PJama$_{1.0}$ Sphere and an ARIES-based log handle this category of failure ($\rightsquigarrow$7.3). These mechanisms may also recover from some hardware failures, but not critical media failures.

In small versions (essentially all stores built with PJama$_{0.0}$ to PJama$_t$) it is feasible to make a (compressed) off-line total copy as backup and to restore it after failure. This kind of stop-the-world archiving was commonly used as back-up to the RVM recovery. It gives the first four versions a high score, in conjunction with RVM's mechanisms. But it is not compatible with the requirement for endurance (G) and it does not scale (D).

The aspirations of PJama$_{1.0}$ to operate at large scales and continuously, require different mechanisms. Firstly, the ability to specify which media to use will allow the log and other store segments to be on specific devices, e.g., a RAID array. A very reliable recovery mechanism based on the log must be available as soon as stores become so large, or operations so continuous, that stop-the-world archiving is infeasible as a backstop [155]. Then an incremental archiving system [158] is required, but it may not be urgent, since, during application development, stores are usually small and operation is not continuous, and hence the strategy of taking a back-up copy can be deployed. Two utilities supporting this are available with the current release: `opjlarchive` / `opjlrestore` giving logical archive and restore ($\rightsquigarrow$9.4) and `opjparchive` / `opjprestore` providing physical archive and restore.

The absence of either evolution technology (E) or migration technology (F) results in stores having to be discarded so that the application can change to meet new requirements. This causes a complete failure of durability, and hence evolution and migration are a prerequisite for durability.

4.2.4 Requirement D — Scalability

As far as possible, application developers should be protected from the effects of scale. The code they develop should operate on hand-held devices (PJama$_{pt}$) and on the largest stores necessary for enterprise applications. Establishing a truly scale-independent programming model is an important research goal.

PJama$_{pt}$ explores the small context and PJama$_{1.0}$ is intended to push into larger scales. The persistent store, Sphere, is designed to handle large store requirements, though the limit with 32-bit PIDs is approximately 100 gigabytes.[8] To explore larger scales would require a new store adapter (Figure 4.1.b) or a 64-bit JVM. Sphere can be recompiled to use 64-bit PIDs. We are currently exploring stores of approximately 10 gigabytes, but we have bioinformatics applications which will need between 2 and 5 terabytes of data ($\rightsquigarrow$12.1 ❼).

4.2.5 Requirement E — Evolution

The moment that application developers have populated a store they discover that changes to the persistent classes and their instances are required. Without a mechanism for handling this the persistence platform is

[8]It depends on the average size of objects as Sphere uses object addressing.

useless, as their only recourse is to rebuild the store using externally held data (i.e., depend on some other persistence mechanism, which is obviously unsatisfactory). The frequent use of schema-editing technology in relational databases attests to the need to support the recurrent changes in application requirements.

Since new requirements on an enterprise application are unpredictable, we aspire to support any change compliant with the JLS ($\rightsquigarrow$8). Such changes typically replace classes, modify the class hierarchy and alter the formats of persistent instances of the classes that are evolving. In the latter case, the format of all corresponding instances must be adjusted, and their new fields assigned values computed in terms of the previous structure. We have chosen to support arbitrary computations to obtain these new values, and have allowed them to be expressed in the Java programming language. To make these computations understandable, the data is accessed from an unperturbed pre-evolution world, into a new post-evolution world. These worlds are then merged atomically after all of the developer transformation code has been executed. Recovery after a crash will either back-off the entire evolution or complete it, depending on when the crash occurred.

As its identifying feature, $\text{PJama}_{0.2}$ introduced evolution, and $\text{PJama}_{1.0}$ now has this facility with a much more scalable implementation. The first version was restricted by the need to construct the two worlds simultaneously in their entirety in main-memory. In contrast, the latest evolution technology operates on a partition at a time ($\rightsquigarrow$7), while maintaining the old-world, new-world illusion and the recovery reassurance. However, user applications still have to be stopped while evolution is run.

Two kinds of evolution requirement may be recognised ($\rightsquigarrow$8): *development evolution* and *deployed evolution*. In development evolution, developers are making repeated, and often frequent, evolutionary changes. In this case scale and endurance are not the most significant issue. We think convenience issues are more significant here; developers are used to tools, such as `CVS` and `make` or integrated interactive development systems, such as `VisualAge` [111], `JBuilder` [29] or `Forte`^{TM} for Java [201]. Developers need equivalent tools for application development on a persistent platform. Integrating evolution and tools remains a research issue (Challenge 5 on Page 75).

Once developers have shipped releases of an application supported by a persistent platform, they will have customers with their own data in stores but requiring bug fixes or upgrades. Deployed evolution is required for shipping these changes and installing them without losing each customer's investment in data and programs (objects and classes). This almost certainly requires incremental installation that avoids interrupting operational processing. Such change installation evolution, compatible with endurance (G), requires research (Challenge 6 on Page 76).

4.2.6 Requirement F — Migration

Migration is a mechanism to allow applications, complete with their data, to move between platforms ($\rightsquigarrow$9). A necessary prerequesite for migration is that PJama is implemented for the target platform. Achieving this has proved labour intensive (see Table A.1 on page 80). The first support for platform migration in the PJama context is an extraction to canonical format, using a stop-the-world archiver, `opjlarchive`, and a restoration from that format, using a similar off-line installer, `opjlrestore`. It has yet to be tried between different platforms and its scalability properties require investigation. In the longer-term, incremental and dynamic mechanisms will be required.

4.2.7 Requirement G — Endurance

Endurance is the ability of the system to run continuously. It may also introduce a concommitant requirement for availability. If we assume that the goal is to run enterprise services continuously, then the platform must run continuously. Apart from reliable code, with no space leaks (on disk or in main memory), we also need all resources to be correctly recycled. The improvements shown in Table 4.1 relate to recycling main-memory data structures. We still need to run the disk garbage collector concurrently, to recycle or archive log space, and to incrementally generate archive copies (these are development goals for PJama$_{1.0}$). Concurrent, large-scale, fault-tolerant and complete disk garbage collection remains a research issue (Challenge 8 on Page 76).

Once an enterprise system is deployed and in production operation, deployed evolution and platform migration *should be* relatively rare. It is probable that not supporting these initially as concurrent, on-line tasks, would not inhibit the deployment for *OP-hypothesis-testing trials*. Stop-the-world versions would be an adequate stop-gap provision.[9] However, their investigation in conjunction with endurance is certainly of long-term significance. Strategies for improving availability include the operation of a "warm" system mimicing the primary system with fast switch over on failure and acceleration of restart and recovery. These are interesting research issues.

4.2.8 Requirement H — Openness

Openness means that the external context of a computation must be accessible without the application becoming inextricably dependent on it. Normally it is expected that this will be via standard interfaces and protocols and that external systems will be able to access the computation in a similar way. As these external agents are not part of the computation they may independently start, stop, fail and change. That is, from the point of view of a persistent platform, they are inherently transient. We observe that handling interaction with such autonomous components correctly is an issue for all long-running systems, whether or not they support persistence.

PJama$_{0.1}$ saw the introduction of Java RMI support. Each release has increased the set of `AWT` classes properly handled (they communicate with that well-known legacy system, "the user", or more accurately the native window system providing communication with a user). It is still not complete for the reasons given above (A). These are all based on our resumable programming model ($\rightsquigarrow$5.5) and registration for platform events [120], which is intended as an underpinning for any external communication. The claim is that, with this technology in place, correctly handling such packages as CORBA and JDBC should be straightforward.

4.2.9 Requirement I — Transactions

Transactions are dealt with implicitly in all versions of PJama except PJama$_t$ ($\rightsquigarrow$10). That is, the start of an execution of the `main` method is an implicit transaction start and termination of the JVM execution without errors is taken as commit, while termination with errors (unhandled exceptions) is treated as transaction abort [120]. This is extended by a checkpoint mechanism that allows part of an execution's work to be made durable; essentially providing a long-running transaction.

It is clear that in an enterprise-server context, many applications against the same store need to run concurrently with guaranteed isolation [54]. Without such mechanisms, typical application programmers are left with two very difficult tasks. First, they have to manage isolation or concurrency correctly with respect to unknown and unanticipated other applications. For this, they would resort to excessive use of explicit locking (e.g., **synchronized** methods) with a resulting poor performance and deadlocks. Second, they would need to arrange that all applications are at a consistent point when they perform a checkpoint. This is probably even more taxing for application developers. Hence there is a high premium on completing a well engineered and integrated version of the model pioneered in PJama$_t$. There are many research issues yet to be resolved in this area ($\rightsquigarrow$10), but the present simple system is surprisingly powerful for a large class of applications.

[9] RDB and OODB users have coped with such limitations.

4.2.10 Requirement J — Performance

Performance is a relative issue. The fundamental assumption of all platforms that raise the level of abstraction is that the resulting gain in productivity is worth the performance penalty compared with expertly hand-crafted low-level techniques. There are rarely enough experts who can outperform a *mature* abstraction; often the work of typical application programmers executes faster when it depends on such abstractions. Occasionally better abstractions allow even the experts to produce faster solutions by using a more sophisticated approach that they would not have attempted with lower-level technology; or permit platform technology to gain performance by optimising the *integrated* load.

It is impossible to second-guess these trade-offs. (There were once plenty of prophets who forecast that relational databases could never provide sufficient performance.) Only by pursuing optimisations and measuring the effect of optimisation on real application loads can we learn what the long-term prognosis is. For the moment, we can see that there is an average of a 15% penalty over similarly implemented JVM without persistence.[10] On the other hand, combinations of database accesses with Java code often runs dramatically slower (factors from 10 to 100 have been observed). Careful measurement and analysis of developed systems is necessary. The present version PJama$_{1.0}$ is a good foundation for such research. Performance is not seen as a gating function by enterprise application architects, whereas availability, reliability, evolvability, openness, access to legacy systems, scalability, and managing the complexity often are.

4.2.11 Summary of Review of Progress towards meeting Requirements

In summary, every aspect of the requirements offers or demands further work; unsurprisingly, as we take a new view of how to support enterprise applications. In many individual cases there are significant research issues, but meeting the requirements in combination is even more challenging. Considering all of the requirements in combination, the PJama$_{1.0}$ platform is a major achievement, has a demonstrated capacity to support large and complex applications, and and is a good platform for major trials.

[10]Though much of this overhead may be optimised away [102].

Chapter 5

An Application Programming Interface

The API should be both minimal and simple. Minimal because larger APIs are unnecessary and impose a comprehension load on application developers. The API defined by the OPJ specification and implemented by PJama is very small because of orthogonality. Once orthogonal support is available, facilities and infrastructure previously implemented by database providers can now be implemented within the language. The most noticable of these are indexes and query languages. If they are implemented within the language, then they are available independently of the platform and can take advantage of a platform's general purpose mechanisms, such as language-based optimization, evolution and migration. Of course, their implementation remains a skilled task ($\rightsquigarrow$12).

Simplicity is a fundamental goal as we want the application developers to encounter no surprises. There should not be exceptions to rules or limits to facilities that they have to remember and work around. The API that was delivered by PJama$_{0.2}$ was simple, but not minimal, and also suffered from having been designed prior to much of the work on the collection class libraries and the new event handling model. Therefore, during the design of PJama$_{1.0}$ we took the opportunity to simplify the API further and aligned it with the developments in the core class libraries. The resulting specification, "Orthogonal Persistence for the Java platform" (OPJ), was formally submitted as a Java Specification Request (JSR) to the Java Community Process and is given in [121]. The OPJ specification permits a wide range of implementations of which PJama$_{1.0}$ is an example. The specification adheres to the established design criteria [22], which are listed below.

Type Orthogonality— Persistence is available to all objects irrespective of their type.

Persistence by Reachability — The lifetime of each object is determined by the existence of at least one path of object references from some defined *persistent root* objects.

Persistence Independence — All code and its semantics is identical irrespective of whether it is operating on short-lived or long-lived objects.

Our experiences in seeking to match these goals with the definition of the Java programming language [91] have been reported in [120] and [18]. Our extensions to the core Java class libraries are defined in [121], which also contains a rationale for the detailed decisions. The highlights of this definition are discussed below.

5.1 Resumable Computation

An underlying philosophy guiding our decisions is the notion of *resumable computation*. If a virtual machine execution were interrupted, its state saved, and then it resumed from where it left off, some time later, the continuation of the computation should be as nearly as possible indistinguishable from the computation that would have happened if the computation had not been suspended. Of course, the computation cannot be guaranteed to be exactly the same, as circumstances may have changed in the interim. In our case, as we are deliberately considering long-term, multi-application computations, the machinery (hardware or virtual machine) may have changed, we call this *migration* ($\leadsto$9), or the developers may have chosen to deliberately change future behaviour, we call this *evolution*[1] ($\leadsto$8). Other changes of circumstance can also occur, as they can during any (particularly long-running) computation. For example, external state managed autonomously may change, e.g., the user at a GUI, the state of a socket, CORBA or Java RMI connections, the schemata of external databases, and the contents of file systems. Our semantics and recommended programming practices will look very similar to the semantics and practices of *any* language designed to support continuous operation, which we call *endurance* — provoking Challenge 4 on Page 75.

5.2 Type Orthogonality

All objects and all primitive types, all arrays of these, and all arbitrarily connected reference graphs of these, are able to persist. Here "all objects" includes system and developer extensions of `java.lang.Object`, including `java.lang.Class`, `java.lang.ClassLoader` and `java.lang.Thread`.[2]

The decision to include class `Class` is a matter of straightforward adherence to the principles; it is an extension of `Object`, and therefore its instances may be referenced in the normal way. We also include it to bring consistency to the interpretation of instances. Logically, there is an implicit reference from every instance to the `Class` instance that describes it and defines its behaviour. If we were to depend on a class being loaded from files, while holding only the state of its instances in the persistent store, we could encounter occasions when the relevant class file was either unavailable or had been changed to be inconsistent with those instances. In other words, our policy makes behaviour as well as state exist. This is very different from the policy adopted by many Java programming language to database bindings. This choice makes it essential that we have good evolution and migration technology ($\leadsto$8 and $\leadsto$9).

A similar argument dictates that instances of class `Thread` should be able to persist. For example, they may be referenced. This commitment is technically more challenging, particularly across all platforms and when evolution and migration are considered ($\leadsto$4.2.1 and Challenge 3 on Page 75).

5.3 Reachability

For the most part, persistence by reachability is straightforward. If an object is persistent, all of the objects that it references must be (made) persistent. Two issues arise: what are the roots of persistence? and what implicit references do we honour?

For the standard Java programming language, the roots of persistence during an execution are the static variables of all classes which are persistent and all active threads (JLS [91]). The persistence of classes is itself determined by the persistence of class loaders and the threads that created them. The notion of persistence during execution is simply a concern for what should not be garbage collected in the `gcheap`. The PJama model extends that naturally to include longer-lived data, providing a definition based on those roots (and optionally additional explicitly defined roots) continuing to exist during a suspension of the JVM's execution. Essentially, this defines which objects may not be garbage collected in the persistent store.

[1]If the Java platform supported modification of a class definition during execution, evolution would be considered to be normal computation.

[2]This is only able to persist in some of our implementations at present. This deficiency is a known implementation bug.

That requires some elaboration. Firstly, application developers, who are typically not language aficionados, may be hazy about how long threads are active or which classes are persistent. Consequently, a mechanism is needed to allow them to introduce persistent roots from which their persistent data structures will persist reliably (`Set roots` in ⇝5.7).

Secondly, we require that computations be self-consistent over resumption. In consequence, classes that were in use before a pause, if they are re-used after that pause, must be guaranteed to be unchanged, unless evolution has been used to deliberately change the future path of the computation (⇝8). Therefore, all classes that have once been used in the computation, and might be used again (e.g. they describe an instance that still persists, or they are symbolically referenced by a persistent class) must be considered reachable. More precisely, if there is a symbolic reference from class `A`, which will persist, to class `B`, and that reference has been resolved (JLS [91]), then `B` should also persist.

5.4 Persistence Independence

Adherence to persistence independence is logically straightforward, though, as already mentioned, it may take significant implementation effort. It is essential, in order that code shipped dynamically from other sources, such as Java-based servlets and Java-based applets, can be run correctly; there are no opportunities to change such code manually. This applies to both byte code and source code forms of a language, and is crucial for allowing re-use. With properly implemented persistence independence, code runs on a standard language's platform and on all persistent platforms unchanged. Such strict adherence to the language semantics is vital for safety (no breaches of the type system or security system) and for code re-use. Conversely, if code were to require explicit calls to a persistence mechanism, then it would only work in the context of *that* persistence platform.

When orthogonal persistence is retrofitted to a language, as in our case, adhering to persistence independence can be irksome. For example, the interpretation of the keyword **transient** in the Java programming language is problematical (requiring workarounds) [176], and we have had to refrain from extending class `Runtime` though that would have been the simplest way to introduce our facilities. We believe, however, that the value of persistence independence is paramount. The long-term solution is to educate language designers, so that these minor irritations are avoided.

5.5 The Resumable Programming Model

Modern applications interact with their external environment, for example, user-interface devices, connections to other (often remote) processes and connections with other systems, such as: name servers, file servers, databases, window managers and web servers. These are intrinsically autonomous, and hence they behave as if they are transient from the viewpoint of the long-term execution model. Typically, they are more likely to change their properties, or become temporarily unavailable, rather than cease to exist altogether.

A program that only executes for a short time may get away with pretending that the state of these external systems does not change. Alternatively, a theoretician, to make a computational model tractable, may postulate some closed universe of discourse with no autonomous objects. With our combined goals of longevity and openness, neither evasion is open to us. A pragmatic solution to this problem was taken by some persistent systems, for example PS-algol [13] and Napier88 [159]. Their implementers understood a specific set of external objects (in those cases: files, an X-windows session and sockets) and attempted to resume the state of the external connection whenever the execution was resumed. Again, this pragmatic solution was not open to us, since the aspiration to support both platform independence and openness means that there is an open-ended set of potential external objects.

Our general solution, called *resumable programming*, gives platform support for application developers who wish to deal with any external subsystem in a manner akin to that adopted by PS-algol and Napier88. During normal execution, the application or its platform holds data structures that represent how it is interacting with external subsystems. These are sufficient for a continuing computation under the assumption of a stable external world appropriate to short-running computations. (They sometimes depend on additional data about these inter-relationships held in the external subsystem.)

In PJama, application developers are able to register that they are dealing with such an external subsystem. Then the persistence platform undertakes to activate their registered handler code, whenever it is about to save the state of the computation and whenever it is about to resume that suspended computation. The developer's handler code uses the opportunity of the first event or call-back to assemble enough information to reconstruct the external subsystem inter-relationship, when the computation later resumes. It may have to make enquiries of the subsystem regarding its current state — for example, the name and position of an open file, the server and socket number of an open socket, etc.

The second kind of event is indicative of a checkpoint immediately followed by suspension of the computation, i.e., a voluntary shut down of the PJVM. It can be used by the application to trigger the release of external resources before the PJVM halts. It would be wasteful to release and restore these on every checkpoint, but it might also be wasteful to keep them beyond a program's execution.

The third kind of event allows the registered application code to use the objects preserved during the previous suspension to attempt to re-activate and reset the connection with the external subsystem. It will need to verify that the required conditions have been reconstructed and may have to admit failure by reporting an exception, or discover an alternative subsystem able to offer an equivalent service.

The specific implementation details of this general strategy for PJama are given in [121]. PJama uses the standard listener and event mechanism for this purpose, via the classes:[3]

`OPRuntimeEvent`, `OPCheckpointListener`, `OPShutdownListener` and `OPResumeListener`.

An `OPRuntimeEvent` simply carries with it which of the three types of platform event has occurred. The three types of listener are activated when the state of the computation is about to be saved, when the PJVM is about to shutdown, and when the PJVM has started up and is resuming a computation.

5.6 Resumable-Programming Style

To take best advantage of this, application programmers need to adopt an appropriate style of programming. At present, programmers often establish the contextual requirements of a class or object during its initialization or creation. Now that these classes and objects are part of a continuing computation they are not re-initialized or recreated when a suspended computation resumes. Therefore an external (or other context) that may have changed can not be re-established via a static initializer or constructor, as these are not re-run.

A resumable programming strategy is to set up methods that establish context, and then ensure that the context is re-established before executing any other methods that depend on the context. The context-establishing methods can be called from initialization code or instance constructors the first time. They are then called again whenever the context needs to be re-established. One method for doing that is to mark some field **transient**. It is then tested to see whether it has reverted to its default value in all methods that require the context. If it has, the context-establishing method is called and it sets it to its useful, non-default value. This is wasteful, as it requires a test at the start of most method calls, but may be necessary defensive programming, if there are possible causes for the context to be lost, which don't signal this with an event. If the context is normally only lost during a suspension of computation, we use the event mechanism described above, the handler for the `OPResumeListener` then calls the context-establishing method.

[3]All of the PJama$_{1.0}$ API classes are in the package `org.opj`.

This model of resumable programming only works if there is no risk of race-conditions with other code trying to resume the same objects. It is difficult to use Java's synchronisation technology to overcome this, as the resumption execution may take place during a restart's bootstrap, when user code is not able to control the computation. This is discussed in [121] and it raises the issue of whether a more incremental and dynamic model of external state restoration is needed. Such a model has been used in parts of PJama$_{1.0}$ [136].

5.7 Summary of the PJama API classes and methods

The simplest API would be obtained if some of the facilities were presented by extension of the existing Java core classes. Since a program (e.g., a conformance test) might perform reflection to enumerate the members of these classes, this would not comply with persistence independence, unless the Java platform itself were revised.

To avoid that issue, all of the PJama facilities are available through the package `org.opj` and all classes begin with `OP`. The relationship with the standard platform classes should be obvious. The *complete* API is listed here to show that it is relatively simple.

```
public class OPRuntime {
    public static void addRuntimeListener(org.opj.OPRuntimeListener listener);
    public static void removeRuntimeListener(org.opj.OPRuntimeListener listener);
    public static int checkpoint() throws org.opj.OPCheckpointException;
    public static int suspend() throws org.opj.OPCheckpointException;
    public static void halt(int rc);
    public static final java.util.Set roots;
}
```

The **static** variable `roots` allows programmers to explicitly introduce their own roots of persistence, such as an index to various structures. The method `checkpoint` causes the computation's state to be preserved, and then the computation continues. The method `suspend` causes the computation's state to be saved for a future resumption and then it terminates the execution of the JVM (unless the security manager prohibits termination). The method `halt`, again only if the security manager permits, terminates the execution of the JVM without preserving the computation, so that resumption will be from the previously recorded state.

```
public class OPTransient {
    public static void mark(
        java.lang.reflect.Field field);
    public static void mark(
        java.lang.Class clazz, java.lang.reflect.Field field);
}
```

This class provides the work around for the present confusion over the interpretation of the keyword **transient** ([176] or [121] for details).

```
public class OPRuntimeEvent extends java.util.EventObject {
    public int getID();
    public static final int CHECKPOINT = 0;
    public static final int SHUTDOWN = 1;
    public static final int RESUME = 2;
}
```

This class is used to indicate which kind of platform event has occurred.

```
public interface OPRuntimeListener extends java.util.Listener {
}

public interface OPCheckpointListener extends org.opj.OPRuntimeListener {
    void checkpoint(org.opj.OPRuntimeEvent event);
}

public interface OPShutdownListener extends org.opj.OPRuntimeListener {
    void shutdown(org.opj.OPRuntimeEvent event);
}

public interface OPResumeListener extends org.opj.OPRuntimeListener {
    void resume(org.opj.OPRuntimeEvent event);
}
```

The preceeding three listeners allow application programmers to adopt the resumable programming style ($\leadsto$5.6).

```
public interface OPCheckpointException extends java.lang.RuntimeException {
}

public interface OPCheckpointInNativeMethodException
            extends java.lang.RuntimeException {
}
```

These two run-time exceptions notify the application of problems encountered during a checkpoint.

5.8 Summary and Issues

The crucial issue for API design is to balance functionality against complexity. The regularity of the PJama semantics, derived from traditional programming language design principles, results in very considerable power. It is not quite minimal, but we believe it is a good compromise, and sufficiently small that it can be assimilated in a few minutes by a program developer.

The main source of complexity is in the facilities for handling intrinsically transient state based on notification of platform events. These facilities are still under debate. For example, they do not support directly the case where there are a very large number of instances of some intrinsically transient class (remote-object stubs for example [195]) and where a typical computation uses only a small subset of these. Another problem is that other activities may be referencing substructures that the restoration code will use when activated. Race conditions or deadlocks may therefore occur, involving the actions of `Resume` listeners. Other schemes for activating resumption code that overcome these problems are being investigated. The drawback is that they are more difficult for application developers to understand, and harder to implement.

Chapter 6

Choice of Architecture

There are a number of architectural choices that must be made when designing an orthogonally persistent platform. As the combinations of choices have not been fully evaluated over *all* of the requirements established for the project (⇝3.2) our choices were guided by intuitions based on past experience, and we are still unable to provide objective measures to justify many of the choices. The major choices were:

- To operate with one virtual machine executing against a single object store rather than allow several virtual machines executing in different processes to operate concurrently against an object store or several persistent object stores to be opened by one virtual machine (⇝6.1).
- To manage a universe of persistent objects rather than manage a persistent address space, persistent page space or persistent segment space that happens to contain objects (⇝6.2).
- To base recovery on ARIES-style logging (⇝6.3).
- To implement by modifying an existing JVM rather than building a system that ran on the Java platform or a totally new PJVM (⇝6.4).

These choices are discussed in the following sections. We then discuss our choices regarding swizzling [164] (⇝6.5) and caching (⇝6.6). Throughout our design, we were trying to balance all ten of the requirements (⇝3.2). This means that if only one group of activities, say *object faulting and promotion* is considered, our choices may appear suboptimal. We often find ourselves having to "lose a little on the swings to avoid a big loss on the roundabouts" to paraphrase a traditional metaphor. The recurrent forces requiring compromise are the need for evolution, for migration, for durability and for main-memory transactions. But the need for solutions that would operate on several platforms was also often significant.

In the discussion below, we will be talking mainly about the released version, PJama$_{1.0}$. There is significant variation between versions and sometimes within versions over architectural choices. For example, PJama$_{1.0}$ has three implementations: a memory-mapped store [145], an experimental log-structured store [145], and Sphere [175]. Sphere is used in the release, as it aspires to meet all of our requirements. The other two do not attempt evolution and disk garbage collection, but are useful for small-scale development work.

6.1 Single Virtual Machine Choice

The most flexible architecture would permit a PJVM to open many stores and a store to be opened by many PJVMs simultaneously. We rejected both of these multiplicities because they introduce significant conceptual, implementation and operational complexities. This choice differentiates the PJama work from many other research and commercial projects. For example, in the OODBMS community, it is virtually de rigueur to choose a model of a server with multiple clients for large-scale stores. In reviewing this decision, we must ask:

- Is it still possible to support the relevant class of applications?
- Is it actually a reduction in complexity?

First, we argue by analogy. Operating systems manage one store, the file space, and one set of concurrent computations, the processes, against that single store, which is equivalent to a *single persistent address space*. They partition and allocate resources to provide the appropriate levels of sharing and isolation for *any* application mix thrown at them.

Second, consider one of the trends in the support of enterprise applications. The effects of Moore's law are that processors become faster and smaller, so that higher performance processors are likely to be physically co-located (on chip, on boards, in racks, and in clusters connected by very high bandwidth networks). In contrast, managing platforms to run applications, becomes increasingly more expensive, per site. Hence it is reasonable to suppose that there will be hardware platforms capable of supporting very large address spaces, very large numbers of concurrent threads, and very large collections of RAM and backing store. This may be manifest as middle-tier business-logic carrying the heavy computation, with thin clients constructed as applets, or it may appear as centralised operating systems with thin client computers, as in Sun Ray^{TM} system [204], or in some other form. But the trend is already evident.

Third, we aspire to build a simple context in which applications may run. If we retain the paraphernalia of multiple processes, different protection mechanisms for main memory and long-term stores, then that aspiration is unachievable. With this context set, we can consider the choice in more detail.

❶ *Locking and Isolation*: Sharing the set of objects available to applications is traditionally managed by locking protocols that implement an isolation policy. All of the individual applications are assumed to be run independently, and to not be communicating with each other directly. Their communication is restricted to writing in the common repository of objects, and when they have finished, others can read. But many human processes are much more dynamically collaborative. If we try to support these on a platform biased towards isolation, the rate at which transactions must interleave becomes ever faster, to simulate shared visibility. Consequently, it becomes interesting to investigate more intimate sharing between the concurrent activities.

❷ *Sharing*: As the complexity of applications that we aspire to support rises: code volume, number of classes, complex algorithms and traversal paths, large numbers and volumes of objects touched, sophisticated interplay between algorithmic and human decisions, multi-threading and intercommunicating subsystems, etc., then the pool of objects is more likely to be shared. Much of this sharing may be described by the programming language, e.g., visibility of methods, rather than some separate abstraction. The operational system should exploit and support users' tendency to focus. For example, a designer using a CAD system, will almost certainly have editing and simulation tools open simultaneously. Similarly a software developer will have an IDE and a build system, and probably many other tools in use. As they move between these tools, they will typically apply them to overlapping sets of objects. But both of these categories of users are likely to work co-operatively. In which case, we expect several users to actively use overlapping sets of objects. Supporting these users, via their own bundles of threads, from one object address space, is therefore likely to be beneficial.

❸ *Protection and Security*: Protection mechanisms are necessary, both to help with localising the damage that can be done by errant code, and as a foundation for security. If a set of applications are sharing a common pool of objects, then that sharing needs to be controlled. We may expect to leverage the protection afforded by strong typing and security managers. But ultimately, an adequate protection system is needed and requires research to develop it. It is needed even without persistence, as multi-developer applications are composed to build bigger applications, the components must neither run amok nor behave as Trojan horses. The FIT developments in the Forest project [59, 65], the Apotram transaction framework [8, 9, 60, 61] and application isolation models [54] are ventures into this territory ($\rightsquigarrow$10). We see no reason, in principle, why protection cannot operate on subsets of an object space, just as it does today on subsets of an address space, on subsets of a collection of rows in relational tables, or on subsets of files.

❹ *Endurance*: As we seek longer-running systems, we need to recycle resources. Shared heaps, shared buffers and shared recovery logging may allow amortisation and load balancing. A major recycling requirement, is to recycle disk space. We do this by using a disk garbage collector [175], which needs to collaborate with the mutator in order to discover which references to structures on disk are currently held in the transient structures of applications. Having all of these applications under one PJVM's management, facilitates this interaction between mutator and collector. Alternative architectures either require more complex distributed protocols or require that the collector makes pessimistic assumptions about what may have been retained in the applications. As we run applications longer, so these assumptions become more costly, as the set of *possibly* retained roots grows.

❺ *Performance*: Assuming the architectural and economic trends described above, we postulate that the cost of communicating between processes will remain *relatively* high, while communication within a process, via the shared memory, will be *relatively* low. Hence we suspect that performance will benefit from avoiding the many inter-process communications that are required by the traditional database-server architecture. A small proportion, but nevertheless a significant absolute number of objects are large,[1] and passing the object identity rather than copying the object saves many expensive main-memory cycles.

A major issue is speed of recovery after failure. Service can resume more rapidly with the incremental object faulting algorithms used in our context. Ineluctably, if these have to be faulted in once per application that needs them in its cache, resumption of service will be delayed longer. Operating with a common pool is therefore expected to accelerate resumption.

❻ *Complexity* We are concerned here with two aspects of complexity, the complexity that faces the application developer and the complexity that faces the persistence platform implementor. The developer concerned with applications for co-operative working or for users who use a set of applications simultaneously (co-operate with themselves!) must visualise and manage an integrated view of the multiple applications. That means that the developer must manage multi-application checkpointing, commits, consistency checking, etc. The partial failure modes of a multi-process system exacerbate this task, but may reduce the frequency of overall failure. Today the end users themselves or some application expert proxy often have to sort out the inconsistencies after failures. Instead, we propose that the underlying platform be responsible for much of this coherence.

The platform developers therefore, take on a difficult task of providing a coherent and stable application environment. If we had to build multi-process co-ordination algorithms, e.g., for lock management, then we believe it would be considerably more difficult. In other words, we are assuming

[1] In one of our applications ($\rightsquigarrow$12.1 ❼) we search DNA sequences. These can be as long as 300,000,000 bases, which we hold in a single array, and index with a structure of about 40 times that size.

that the hardware platform, and perhaps a minimal kernel operating system, will provide a low-level, coherent integration of a set of processors, a main-memory and a set of disks, even though they are intrinsically a network of autonomous units.

The set of possible implementation architectures is immense and scarcely explored. The many components involved are individually complex and their properties continue to change relative to one another. It is unfortunate that we continue to make our decisions based on "engineering intuition" rather than "engineering theory" firmly founded on measurement. Because we have opened up a new region, we have a responsibility to capitalise on that through well-formed experiments to initiate those measurements. All we can say so far, is that we haven't found evidence to refute this choice, and have some evidence to support it (⇝12). The choice of architecture remains an open question and, although there are some beginnings [54, 59], the choice of multiple applications within one machine is under explored.

6.2 Choosing to Manage Objects

We choose to manage objects rather than pages mainly on logical grounds. Programs manipulate objects. We hypothesise therefore that it is preferrable to perform as much of the management as possible at the object level, since the interaction between the executing program and the objects is much more frequent than our interaction with devices such as disks, which force a different unit of management.

When we consider space management, the object management technology, e.g., on the `gcheap`, is already well developed. Our additions to it, build on a framework that is the subject of significant research effort [219].

Objects vary widely in size, but are very unevenly distributed. Typically, we see an average size of from 40 to 80 bytes with a few very large objects and a substantial population of small objects. These latter are often the fabric of navigational structures, tree nodes, list nodes, etc. Algorithms usually traverse a sparse subset of these, often repeatedly, determined by the current "focus of interest". As they are much smaller than pages, it is more likely to be possible to house their current working set in memory. Indeed, the incremental faulting models on which we depend are based on the observation that our applications fault in the objects they need and then use them repeatedly. This contrasts with the behaviour of simple queries.

Present hardware platforms have memory-management and protection hardware tuned to a flat-address-space processing load. Typically it assumes page-based space and protection administration. It is conceivable that if that structure continues, the mismatch between object management and the hardware will have a high penalty. However, object-oriented (or other information hiding) programming languages with fine grained structure and automated space management appear to be here to stay and central to commercial activity. So we may expect to see the hardware develop to better support object management, just as it did to facilitate image processing. For example, support for sufficiently fine-grained read and write barriers, to facilitate incremental garbage collection, may appear and be general purpose, so that it can facilitate object faulting, object locking, update logging and object protection. Similarly, a compare-and-set instruction that operates atomically on one bit would allow more compact multi-thread safe administrative structures.

Once again, measurement-based evidence on which to base such decisions is highly desirable, and as yet unavailable for full-scale loads.

6.3 Choice of Recovery Mechanism

This choice is partly dictated by the preceding choices. Essentially page-based systems can use page logging or shadow paging, but these are not appropriate once objects are the unit of management. We chose ARIES-logging (⇝7.3) because it is possible to achieve massive reduction in log traffic by taking advantage of logical logging. This is a significant advantage when performing bulk loads [177] and evolution (⇝8).

6.4 Choice of Implementation Strategy

We have chosen to modify an existing JVM (various versions of the JDK and for PJama$_{1.0}$ the SRVM[2] derived from the JDK). The alternative would have been to work with a different JVM (this was not an option when we started).[3] So the only real alternative initially was to build our own from scratch. Working with an existing JVM has had three drawbacks.

❶ It is complex and difficult to work with (partly because all the versions we have used are derived from the original experimental code base).

❷ It is sometimes targeted at different anticipated loads than those we aspire to, for example, 64-bit addressing would match our scale requirements, and sophisticated JIT compilers would be worthwhile in the long-running server context.

❸ It is not a good foundation for research experiments; for example, its data structures are not in the Java programming language, and so we cannot easily preserve them to exploit persistence in the cause of better optimisation.

Starting afresh would involve substantial work. The abstract machine for the Java programming language is complex, and hence any interpreter or code generator would need a significant amount of work. It would be sensible, as others have, to minimise the complexity by focusing only on the functionality needed on the enterprise servers. This omits the complexities of AWT peers, etc., but it would still require that an application's GUI data structures, probably composed of Swing components, could be stored. It still requires external state for CORBA, JDBC, Java RMI and sockets to be properly handled.

Given our interest in long-running and large-scale applications, a sophisticated optimising compiler would be essential. It would be written in the Java programming language, so that it could benefit from persistence, e.g., to store data-flow analysis graphs [222]. It could perform optimisation during quiescent periods or by using opportunistic access to stores, improving their performance for the next time that the load rises, based on long-term usage statistics, and data population analyses. While such an approach looks very attractive now, it was infeasible when we started, as it depends on an orthogonally persistent Java platform for its development.

Though any JVM we built ourselves would soon be complex, it could adhere to coding policies and technologies that are persistence-friendly. For example, all core classes and implementation classes would have appropriate resumption properties and externally connected classes would not depend on establishing their context during class initialization or instance creation. Similarly, the `Thread` implemention would provide enough introspection to snapshot and restore thread state.

If such a strategy were used by a research project today, it would be tempted to clean up some language infelicities, such as misuse of **transient**. That is, when dealing with source code, to admit an "idealised" Java programming language variant, suitable for systems research, just as the theoreticians have an "idealised"

[2]Reminder: Sun Labs Research Virtual Machine.

[3]But both Kaffe (www.kaffe.org) with its open source and wide spectrum of available platforms or Jalapeño [6] as it is written in the Java programming language and has reasonable performance, would certainly be considered as candidates now, although Jalepeño is not publically available. The candidate of choice at present might be Open-JIT [170], which would probably need to be adapted to a suitable run-time platform. We certainly require a good quality JIT [199], but with access to optimisation data structures and processes.

Java programming language suitable for formal reasoning [112]. This raises the interesting question as to precisely what form that language should take? Taking this further, is the semantics of the virtual machine sacrosanct? Should we investigate a different semantics for threads, where they only stop at safe points, or a different semantics for class loading and resolution that simplifies ambiguities over the timing of resolution and initialization? It is likely that such excursions would only expose different problems and reduce the ease of finding useful workloads and hence of making comparisons based on relevant measurements.

6.5 Choice of Swizzling Strategy

The choice of object management permits, but does not require, PIDs to have a significantly different structure from virtual-memory addresses. PIDs could have been byte offsets inside the store. They are not, so that incremental and locally determined disk-space management can be achieved [175]. Another reason is that it permits localised rearrangements when objects change size during evolution, and because the logical-to-physical partition mapping makes it easy to atomically introduce a rearranged partition ($\leadsto$7). In principle, this also permits us to use PIDs that are larger than the virtual-memory addresses used by the PJVM, but we do not exploit that opportunity at present.

Consequently a swizzling operation is needed before a pointer in an object is used by the PJVM and an unswizzling operation is needed before an object containing a pointer can be written back to the persistent object store. This has been a requirement, since the earliest persistent programming language implementations [47, 15]. Many options are available about when to swizzle, from never (look up in the resident object table (ROT) for every dereference) to eagerly, but incompletely, as in PJama$_{0.0}$ [62]. The choice depends on the relative cost of barriers, lookups in the ROT, updates to objects, on whether there is an indirection in references to objects, and on the workload, particularly the frequency of multiple dereferences, and is much discussed in the literature [163, 225, 221, 220, 125, 149].

We have chosen to arrange that the representation in the disk buffers is always in disk format (i.e., unswizzled). When an object is copied onto the `gcheap` certain references, e.g., to the class of an instance are swizzled immediately, which may result in faulting in the class object. Pointers are swizzled before they are placed onto an evaluation stack. This could be redundant if the pointer is merely being copied, but this avoids a read barrier to arrange object faulting, whenever a method variable is dereferenced [136]. A JIT could optimise this.

However, the variables on the stack can hold references for long periods to persistent objects that are not in use. In PJama$_{0.2}$, we experimented with a scheme which permitted such (non-mutated) objects to be evicted and re-instated on stack retraction. This proved too large an overhead, once we had JIT compilation, and so, in PJama$_{1.0}$ objects directly referenced from any thread stack are unevictable. Any direct use of pointers in an object will cause them to be swizzled if necessary, but the `clone` and `System.arrayCopy` methods do not cause swizzling.

When to swizzle is another design space that would benefit from up-to-date measurements derived from running real loads.

6.6 Caching Choices

In the early versions of PJama, PJama$_{0.0}$ to PJama$_{0.2}$, the active persistent objects were cached in a persistent object cache and the transient objects were allocated on the `gcheap`. This proved complicated [62] at least in part because the virtual machine did not have exact pointer information, and it was particularly limiting during promotion,[4] as a copy of every newly promoted object had to exist in both the persistent

[4]The process of transferring newly persistent objects from the `gcheap` to the persistent store.

object cache and the gcheap. This had a high space and copying overhead, and because promotion was then not incremental, this limited the maximum volume of promotions during a checkpoint.[5] Consequently, in PJama$_{1.0}$ the objects that are faulted in are allocated space on the gcheap. This means that the garbage collector must become responsible for causing evictions. Measurements show that a second-chance algorithm currently gives the best performance [136]. The problem with integrating the object cache and the gcheap is that it currently leads to a complex relationship between main-memory space management and object-cache management algorithms, though there is potential for synergy — see Challenge 15 on Page 78.

PJama$_{0.0}$ to PJama$_{0.2}$ and PJama$_{1.0}$ have all used a separate page cache for disk pages. Recent measurements have shown that increasing the size of transfers between disk and this cache significantly improves throughput [177].

At present no computation by bytecodes is performed on objects in the disk cache. Instead, they are copied onto the gcheap first. For objects that are only used once (or a few times before the page has been evicted) this is extremely wasteful of main-memory cycles copying the object. So far we have not assayed the strategy of copying to the gcheap only if the page is being evicted and there is evidence of further use, mainly because of the complexity it introduces and because of the cost of gaining that evidence. The complexity arises from the increased coupling between the virtual machine and the store layer and because object-references may have been constructed to the object in its present position. This indicates further research into store APIs is warranted. But copy on eviction should surely be investigated, particularly for large scalar arrays, which are often scanned only once (see Challenge 15 on Page 78).

Another issue worth investigating is trading between disk cache and gcheap space dynamically, as different applications make quite different demands. Some collections of data admit of substantial compression, which accelerates applications by avoiding transfers [49]. Sphere would accommodate regimes that compressed data, but again, this has yet to be explored.

6.7 Summary of Architectural Discussion

The choice of architecture for a persistent programming platform remains a difficult issue. We would not expect one architecture to suit all contexts. For example, a static RAM mechanism is appropriate for PJama$_{pt}$ on a personal data organiser [146], a shadow paging mechanism or log-based store may be convenient for medium-scale and short duration development work, but the full panoply of Sphere and PJVM may still need further elaboration for continuous and large-scale operation. The tradeoff between code complexity and performance is also subtle, so it is not always worth attempting every optimisation.

Our goal is a system that is simple to use. This includes one which is simple to set up and simple to start an application running in it. The first requirement militates against configuration parameters, and yet we have found ourselves forced to introduce a configuration file for describing stores that span multiple files and partitions. The second requirement militates against a multitude of parameters on the command line. But we find ourselves with an embarrassing number of these parameters. If the application load really runs for very long periods, it will be impossible to characterise its behaviour with a single set of these parameters. So, to achieve a “just press the start button” modus operandi, we need to automate the discovery and adjustment of the operational parameters. An example that suggests this goal is achievable is AutoRAID [89, 223].

Both the predictable and understable design of persistent platforms and their “knobless operation” require better models calibrated by real engineering studies, which presents Challenge 9 on Page 76.

[5]This is not a fundamental limitation consequent on choosing to separate the object cache from the gcheap. If the VM provided exact information about pointers, it would be possible to use both spaces incrementally.

Chapter 7

Overview of Store Design

The principal persistent object store underpinning PJama$_{1.0}$ is Sphere, which is definitively described in [175]. This store was designed to accommodate a range of applications, including a succession of virtual machines, but with formats for objects closely coupled with the requirements of whatever virtual machine was currently using it. This controlled coupling with the imposed *mutator* load is described in ⇝7.1.

The design of Sphere assumes the architectural decisions described in ⇝6 and aspires to satisfy the storage aspects of all the requirements (⇝3.2). These include supporting orthogonality by being general purpose (⇝7.2), coping with a variety of loads with reasonable performance (⇝7.5), supporting durability in conjunction with the recovery mechanism (⇝7.3), supporting incremental store-management algorithms and thereby endurance (⇝7.2), supporting evolution through an indirection mechanism (⇝7.4) and supporting migration through format descriptors (⇝7.1). The choices made within the store design are best described in [175]. (Earlier discussions on aspects of Sphere are in [174, 179, 180, 178].) The following sections simply discuss the highlights.

7.1 Managing the Relationship between a Store and its Mutator

We refer to the application that is using Sphere as the *mutator*. A mutator may be some C or C++ application, or an application in some other language and the run-time context and virtual machine of that language. The latter is the case when we consider PJama. One essential requirement on Sphere was to accommodate such a variety of mutators without requiring excessively expensive mappings between a mutator's representation of objects and Sphere's stored form. This flexibility is important even within the context of PJama as we foresee a succession of PJVM implementations. This balance between efficiency and flexibility is achieved via two constructs.

❶ A variety of *kinds* are supported, so that the mutator can find a storage format that matches its own requirements.

❷ *Descriptors* are introduced to describe the mutator's use of some flexible kinds.

The range of kinds accommodates repetition structures, such as arrays of each base type, and aggregation structures, such as object instances consisting of a regular structure of elements that is determined by that object's `Class` and hence needs a descriptor. For the most part, this permits the storage format to reflect the mutator's format, so that images of objects can be copied easily from the page buffers into the object cache on the `gcheap`. Exceptions to this bit-wise copy are handled by a call-back mechanism, described in ⇝7.2.

However, Sphere has to manage each object throughout that object's life-cycle. This life-cycle, *as viewed by Sphere*, comprises:

❶ *Object creation* — during a checkpoint, allocate its PID and space, and then copy it to disk [177],

❷ *Object faulting* — given a PID, locate its object and transfer it, via the disk cache to the execution context,

❸ *Object update* — during checkpoint or eviction,[1] write a log record (⇝7.3) and write the new value over the stored image of the object on disk,

❹ *Object migration* — movement of the image of the object onto a new platform,

❺ *Object evolution* — during evolution, transform the object's format, and

❻ *Object death* — detect that an object can no longer be used and recycle its space and PID.

These first and last steps occur only once in each object's lifetime. Other steps may occur in any order and any number of times during an object's life, except that object update must always be preceded by object creation or faulting. To support these steps in an object's life, Sphere has to know enough about each object: for example, which fields are pointers (to support migration, evolution and detection of unreachability via disk garbage collection) and how large each scalar field is (to support migration and evolution). One strategy for obtaining such information is to make up-calls to the mutator requesting each bit of information each time it is needed. This was rejected for two reasons.

- It requires the mutator to be always available, whereas, operations such as disk garbage collection and migration may be scheduled when the mutator is not available.
- It places a dependency on the mutator being consistent and efficient in returning this information and tends to result in complex relationships between the store and mutators, and may increase the execution overheads.

Another strategy is to force the transformation of every object image into a store-specified canonical format. This introduces translation costs every time an object is written to store and read from it. Although Sphere would support such a model of working, we did not want to impose it permanently on all store-mutator combinations.

The solution was to introduce *descriptors*. Whenever a kind is not simply self-describing, Sphere requires a descriptor. A descriptor must be supplied by the mutator when an object is created. It contains an abstraction of the mutator's choice of format for that object. Objects with the same format share the same descriptor. The interpretation of descriptors is specified by Sphere. Thus Sphere is assured of available and consistent information about each object and it does not depend on the continued availablity of the mutator. It stores and manages each object throughout its lifetime, according to its unchanging descriptor, or its kind.

7.2 Overview of Sphere's Organisation

Chapter 3 of [175] provides a definitive account of the design of Sphere so only a synopsis is offered here. Sphere takes a very general view of what an object is. It is any sequence of fields, where a field can be a scalar of 1, 2, 4 or 8 bytes or a pointer of 4 bytes.[2] Objects may be large, up to several hundred megabytes, though we expect many small objects.

Each object has a *kind* and *persistent identifier* (PID), and for some kinds they also have a reference to their *descriptor*. Objects are grouped into *partitions*, which contain a header, and a number of objects

[1]At present we do not evict mutated objects.

[2]Sphere can be recompiled for 64-bit pointers, but cannot operate with a mixture of address sizes.

referenced indirectly by an *indirectory*. A PID consists of a pair `(LP#, E#)` where `LP#` is a *logical partition number* and `E#` is an *entry number* which is interpreted as an offset in the indirectory.

Partitions are introduced to permit incremental store management algorithms, such as evolution ($\rightsquigarrow$7.4) and disk garbage collection (Chapter 6 of [175]). A mapping between physical and logical partitions is used to permit simple atomic transitions from an old to a new form of a partition, as in Challis' algorithm [43, 44] or shadow paging [11, 45, 166].

To permit use of multiple files and disk partitions, Sphere introduces *segments*, and uses them as the source of space for a two-level space administration system. Each segment is a sequence of *basic blocks*. When a partition is allocated, a contiguous sequence of basic blocks of sufficient size is found. Objects are allocated within partitions. This allows object allocation to take place concurrently.

Sphere manages a set of disk buffers and operates a *steal* policy, that is dirty pages can be written back to their home site on disk before a transaction (checkpoint) commits. This is combined with a strategy that minimises log writes when new data is written to disk [177] or Chapter 5 of [175]. This combination allows very large volumes of data to be added to the store efficiently during one checkpoint.

Sphere provides two disk garbage collectors at present. One processes individual partitions and the other, `opjgc`, which is normally activated explicitly by users, processes an entire store in order to remove multi-partition garbage cycles. At present, they both require that the mutator be stopped while they operate, which is not compatible with our endurance requirement. An experimental version of a disk garbage collector that operates concurrently with the mutator will be converted into a component of the released system in the near future (Chapter 6 of [175]).

7.3 Recovery Mechanisms

The recovery mechanism is based on the ARIES write-ahead logging algorithms [157, 156, 155] so that we can exploit logical logging and other techniques for using relatively small amounts of data to support durability. An introduction to ARIES, our rationale in choosing it, and a description of our implementation and its use to support Sphere and PJama$_{1.0}$ requirements can be found in [95]. The most frequent exploitation of logical logging is recording updates to inter-partition cross-reference counts. The most significant saving is obtained from the succinct recording of partition allocations, of space allocation and of PID allocation during a checkpoint [177], and the succinct encoding of garbage collection and evolution progress as modifications to the logical-to-physical partition map [175, 96]. Durability is augmented by two stop-the-world archiving programs in PJama$_{1.0}$:

- A physical archiving system, `opjparchive` and `opjprestore`, which operates at the store level and linearly traverses all of the disk segments in use, making an optionally compressed copy. This can be restored only into the same context and will only operate with exactly the same version of the system.
- A logical archiving system, `opjlarchive` and `opjlrestore`, which is written in the Java programming language and uses reflection to extract the entire state of the Java application(s) ($\rightsquigarrow$9.4). The standard `java.lang.reflect` reflection needs augmentation in order to deal with all of the implementation classes, such as class loaders, `Runtime`, `System`, etc. It has to deal with these in a special way, as they must be *unresolved* and their C-state abstracted, so that restoration is possible with a new version of the platform that uses different versions of the implementation classes.

It is unsatisfactory to only have stop-the-world archiving, and an incremental binary archiver should be implemented [158]. There is an interesting interaction here with logical logging. For example, the exploitation of fast first writes during new-object promotion [177] means that images of the new object values are not written to the log. So the traditional incremental archiving algorithms, which collect the new data from the log, would not see these new values. But writing them to the log would lose the acceleration and log capacity afforded by treating first writes specially. In fact that is not necessary, as the log contains the range of new PIDs and hence the range of new pages and partitions allocated. Hence the log can provide indirect information and the incremental archiver can perform a linear forward read from those disk areas, as logical operations are not applied to the data areas of a partition.

As we address the issues of scale and endurance, we will need to improve the recovery technology to include:

➊ More succinct update records, tracking the differences in partial object updates at a finer grain.[3]

➋ Adaptive logging algorithms, extending the space for the log or its cache, for example.

➌ Better log-space recycling. At present the log on disk is simply treated as a cyclic buffer and if the head gets too close to the tail, the current execution is forcibly stopped and rolled back. If some virtual logs refer to terminated transactions, their data is no longer needed. It is therefore possible to copy the active log records, omitting the the redundant data, and save log space.

In summary, we are satisfied with our choice of ARIES as the foundation for our recovery technology, and it does seem to work well, but there are still improvements that would gain performance and functionality that we can make.

7.4 Sphere's Support for Evolution

The technology to support evolution ($\rightsquigarrow$8) that has been integrated into Sphere is best described in [96]. The engineering challenge that the store technology has to meet is described as follows.

Evolution operations must be atomic, that is in the event of a crash the evolution either completes or the store reverts to its original state on restart. An evolution may alter very large numbers of objects, and we need to avoid generating excessive volumes of logging data or demanding very large main-memory spaces.

The evolution cost should be approximately linear in the number of objects that must be reformatted. This should be true whether a very small proportion of the persistent objects, or the majority of them, require reformatting. We also require that the execution be fast, which is a matter of ensuring that the predominant operation is scanning the disk space forwards linearly, skipping space that could not hold objects requiring reformatting.

The transformation applied to each evolving object is either a default, or is code specified by the developer. In order that this code can be written easily, we need to continue to present the pre-evolution state consistently to executions of the Java application throughout the evolution. At the completion of applying all of these transformations, we must switch the store state to present only the post-evolution state. A two space algorithm is infeasible because of the size of the store and because often a minute fraction of the objects are transformed. Object identity must be preserved, even when objects change size during transformation. The mechanisms used to achieve this can be summarised as follows.

- The use of a partition and regime structure to selectively scan the store linearly and incrementally.
- The use of a descriptor invariant to rapidly discover whether any instances of a class exist within a partition and hence to rapidly identify the extent of the classes to be transformed, without having to

[3]A first version of this reduced the log size by 40% for one index-building bioinformatics application.

maintain class extents.[4]

- The use of the disk garbage collector's algorithms to manage identity and copying.
- The use of the logical-to-physical partition number mapping to achieve durable atomic transitions without excessive log traffic.
- The use of hidden *limbo* versions of objects to:
  - Avoid log traffic.
  - Avoid PID allocation and maintenenace of an old-PID, new-PID map.
  - Hold the old and new state simultaneously at a cost proportional to the actual changes.
  - To reveal the new state atomically at the conclusion of evolution.

At present this appears to scale well provided that the developer leaves it to our evolution technology to iterate over the classes to be transformed [17]. We have been able to apply both default and developer specified transformations to stores containing 60 million objects that had to be transformed. When we attempt incremental evolution algorithms that run concurrently with the regular operational load, then the support in Sphere will have to be enhanced.

7.5 Review of Sphere's Performance Potential

The integration of Sphere with a PJVM is only recent, so the only available measurements for Sphere that investigate performance are reported in [175]. Those for bulk-update are extensive, but the rest can only be considered as preliminary analysis of the sensitivity of performance to various design parameters, as C programs were used to synthesise the load. In all cases they are based on micro-benchmarks, because they were specifically targeted and because full application loads were not available.

These measurements consistently show the advantage of reading and writing larger units and the considerable increase in throughput from concurrency. These findings have yet to be exploited in our released platform. We suspect that more acceleration can be obtained by increasing the use of asynchronous transfers. Large effects using compression have been observed elsewhere [49, 224] and their value in this context should be investigated. The Sphere store already supports highly parallel use by the mutator, but this has not yet been exploited to improve performance when using multiple disks. The regime-structure permits hybrid systems which might use such techniques where they are effective.

7.6 Summary of Sphere's Status and Future

The present version of Sphere has been used with 5 GBytes of data and 60 million objects. We expect it to scale comfortably to somewhere between 25 and 100 GBytes (depending on object size) with 32-bit PIDs and to operate at a much larger scale with 64-bit PIDs. The largest object we have observed an application place in the store, so far, was a byte array of 34,000,000 elements holding the DNA sequence for human chromosome 22. The present work with bioinformatics applications will push the 32-bit PID version to its

[4]Sphere arranges that if there is an instance in a partition, the descriptor that describes that instance is in that partition, so that partitions can be processed independently. If the disk garbage collector eradicates the last object associated with a descriptor from a partition, it also removes its descriptor. There is an index of the descriptors in each partition in its partition header. Hence by checking whether the relevant descriptor is in that index one can rapidly determine whether there is an instance of a particular class anywhere in the partition without reading and scanning the partition.

limits (⇝12.1 ❼). Recovery and archiving combine to give very good reliability *for a research prototype*. The Sphere framework supports incremental garbage collection, which is currently being incorporated into the main platform. Though Sphere is general purpose, it integrates well with a PJVM. Much investment has been made in supporting incremental, safe and durable class evolution (⇝8). Immediate benefits can be obtained from tuning [175], such as significantly increasing transfer unit sizes.

The infrastructure provided by Sphere and the access to real workloads provided by PJama$_{1.0}$ combine to present an unprecedented opportunity for persistent object store research. Researchers can now explore:

❶ Specialised storage schemes, e.g., compression, using the regime structure.

❷ Hybrid space administration on a per-regime basis.

❸ Advances in disk store garbage collection, including cycle collection and parallelisation.

❹ Observing object-lifetime, access and update patterns, analysis of which may lead to insights into store design, short-term tuning and longer term dynamic, self-tuning operations.

❺ Advanced composite store management algorithms, such as integration of incremental garbage collection with reclustering, evolution and archiving.

As we target larger stores (tens of terabytes), using many disks and complex storage devices [223] the use of more parallel-processing algorithms becomes absolutely essential. An important direction for persistent object store research is therefore a highly parallelised Sphere. Ineluctably, major architectural changes will be needed, but Sphere will allow us to approach scales which allow informed design of those new architectures. Our current store research therefore places us in a position to tackle Challenges 10 and 11 on Page 77.

Chapter 8

Schema Evolution and Restructuring

Without doubt, an effective evolution technology is essential in any persistent platform. One of the dominant effects of time is change, and in enterprise applications, this manifests as changes in requirements, changes in understanding of the application and changes to correct mistakes [192]. In the case of an object-oriented persistent platform, these changes result in a requirement to change the descriptions of existing objects, both their content and behaviour, to change those objects to conform with the new descriptions, and to introduce new descriptions that will later generate new objects.

Zicari explored the principles of this issue in the context of O_2 [229, 41] but was unable to complete the implementation of his ideal models because of restricted access to the O_2 platform. Odberg identified a classification of object-oriented schema evolution techniques [169] and Wegner and Zdonik have examined the issues of schema edits in OODBs [228, 217]. In all of these cases, it was not possible to verify that the specifications of behaviour were consistent, i.e., they were unable to analyse the code describing behaviour because a binding to it was not included in their stores. As far as we know, this is also true of commercial RDBMs and OODBMs that support schema edit, such as Gemstone. We describe our approach to schema editing ($\leadsto$8.3), covering:

❶ The way that changes are specified.

❷ The set of changes supported.

❸ The consistency checking undertaken.

❹ The set of object population reformattings that may ensue from such a change.

We discuss some of the implementation issues encountered. There are two important contexts in which evolution is required: *development evolution* ($\leadsto$8.1) and *deployed evolution* ($\leadsto$8.2).

8.1 Development Evolution

When developers are developing a persistent application they will frequently want to test it against persistent data. They therefore work on a collection of data and classes that represents the currently relevant aspects of the eventual enterprise application. At this time, they do not usually require more than a few hundred megabytes of data to get representative behaviour and run all of the tests. For example, when testing the geographic application of persistence (GAP), we use a few counties of the UK or of California, rather than the whole UK or USA data ($\leadsto$12.1 ❶).

It is typical of this phase of use, which may include user trials, that changes occur very frequently. The developer then needs to change classes and install them as replacements in the experimental or prototype store. As the store is small, this can be accomplished in a reasonable time. As any users involved are aware that this is a development system, it is acceptable to interrupt the prototype service if one exists. Normally, using the evolution technology proves much faster than rebuilding the store. However, we have observed developers still rebuilding the store and conclude that convenience is crucial to the use of evolution technology in this context. We are therefore working on integrating it with build and compilation technology (⇝8.4).

8.2 Deployed Evolution

Once developers ship a version of an enterprise system we expect it to be in constant use at a large number of customer sites. These customers, or other bespoke software vendors, will develop their own software (classes) and populate their stores with the shipped classes, their own classes and instances of both sets of classes. Meanwhile, the developers have fixed bugs or provided new facilities and need to ship the revised classes to the stores of customers that want the bug fix or new release. Several issues must now be dealt with.

❶ *Validation and Preparation* — In the case of development evolution we have the target store at the site of development. Hence we are able to run validations that inspect the classes in that store and the instance populations. It is trivial, for example, to evolve a class that has no instances. This inspection is no longer possible for deployed evolution. The evolution technology must therefore be coupled with configuration management, so that validation and preparation can be based on what has been previously shipped to the customers' stores. In this case it will be necessary to be conservative regarding the changes to external interfaces that customers may have used and so the evolution system should support identification of commitments that should not normally be changed.

❷ *Defining the Transformations* — The developer of a collection of classes now becomes responsible for technology to move the computation state forward, transferring information with appropriate transformation code, that itself must be robust and capable of executing concurrently in a customer's environment. This is equivalent to the code that handles versions of proprietary representations in word-processing and graphics packages, except that it has to work incrementally on a graph of objects that may be in use. As in these examples, the code may encounter instances of even earlier versions because customers do not always install upgrades.

❸ *Packaging the Change* — The set of changes to the class population and the set of transformations has to be assembled into a package and transmitted to a customer. This self-contained evolution recipe has to be easily initiated and has to be controllable. For example, customers must be able to express their choices as to how eagerly the changes are applied and must be able to control the resources used to the point of terminating progress with the installation. Problems need to be isolated, so that they do not damage a customer's data or operations and so that they can be reported to the developers. This evolution package has to verify that its assumptions are met before it allows itself to be applied. The customer may choose to make its application partial, so that, some of the "applications" in the store may be upgraded and some may remain in their older version.

❹ *Applying the Change* — Once the installation of the evolution is initiated it must operate concurrently with a customer's operational work load, as non-disruptively as possible, i.e., it should be incremental and may be lazy. This requires that the old and new model can co-exist in the running computation and in the store, and that they are safely separated so that they do not destructively interfere and so that the evolver does not deny the operational system services. At the same time, the normal requirements for durability and rapid recovery after a failure must be maintained. We believe that fine-grained transactional isolation ($\leadsto$10) is needed to support these requirements.

All four of the above issues still require research before deployed evolution becomes available. We suspect that it may be possible to extend the descriptor mechanisms to have a sequence of versions of the class co-existing and that it may be possible to couch the concurrent incremental application of the evolution as a long running transaction that progresses opportunistically on one partition at a time. Investigating methods of delivering deployed evolution leads to Challenge 6 on Page 76.

8.3 Class Evolution in PJama$_{1.0}$

The current class evolution mechanism in PJama$_{1.0}$ is only suitable for development evolution, since it requires that the execution load be stopped while the complete process is applied eagerly to the whole store. However, it is scalable, it carries out consistency checks, runs to completion or restores the store to its original state, and is completely general. The main steps in an evolution are described below and more fully in [96, 77]; background and evaluation can be found in [76, 78, 17] (the last of these includes performance measurements).

❶ *Specify the Set of Class Changes and Instance Transformations* — The developer generates, or obtains in the case of third-party code, a set of revised classes and asks the evolution tool, `opjb`, to install them in a specified store. Typically this will mostly be changes to existing classes, produced by editing their source, and new classes. Some of these changes may change the class hierarchy, and specific renaming, deletion and insertion of classes is also possible. This approach allows the developer to specify the new versions using familiar tools. It also clearly supports any possible change to the set of classes in use with the store.

The developer must also define methods (see the cited references and user documentation [172] for details) to take information from old instances of a class to new instances of that class or some related class whenever the default transformations will not suffice. These transformations are specified in the Java programming language and use a systematic renaming of old classes, so that both the old version and new version can be used in the same code. Typically these methods are supplied with a reference to the old instance and a reference to the new instance which will already contain the results of the default transformations. This code then reads data from the old instance and may carry out traversals of any data reachable in the old world and generates new data, possibly including completely new data structures, and assigns them to the new instance. We believe that this permits any possible transformation between old and new data structures.

In order that this developer-supplied code should have comprehensible semantics it is essential that it should see each "world" consistently, i.e., following references from the old instance traverses the old object graph undisturbed. References from the new instance may refer back to the old world (as yet untransformed instances or instances of classes that are not changing) or they may refer to wholly new structures visible only from the new world. There are many challenges to achieving this in a scalable way [96].

❷ *Validate, Complete and Analyse the Set of Changes* — A specialised and extended version of the standard Java compiler is used together with additional analyses to verify that:

- The set of changes is complete.

- The final state will comply with the semantics of the Java programming language as written in the JLS [91].

The first is achieved by maintaining records of the classes used with and contained in the store and comparing those with the classes presented or reachable through the current `CLASSPATH`. The second is achieved by examining the full set of resulting classes, by compiling them together and including in the analysis all of the referenced classes that are already stored persistently in the store. For example, this kind of check ensures that if, after evolution, class `A` remains in the store and it uses method `m` and variable `v` of class `B`, then `B` still has members `m` and `v`, and `m` has the expected signature, and `v` has the expected type. If any inconsistency in the target state is detected, it is reported and the evolution operation is abandoned with the store unchanged.

An outcome of this analysis will be a set of class replacements, R, of the form $\mathtt{C} \mapsto \mathtt{C}'$, a set of new classes, N, to be inserted and a set of classes to be deleted, D, and a set of transformations, T, (some developer supplied and some default). For each member of R there must be a corresponding member of T or there must be no instances of the class currently in the store. This latter property can only be verified by a scan of the appropriate partitions of the store, as we do not maintain extents of classes, due to the overhead this would impose on normal operation (⇝7.4). Similarly, for each class in D, there must either be no instance of that class in the store or there must be a method in T that assigns and transforms its instances to other classes. Further analysis of R yields a set of cases where there is a changed format for the instances, CFR, where $CFR \subseteq R$.

We enter the next phase with a set, CV, which is all of the classes that must be visited during store traversal. That is, all of the old classes `C` in CFR and any others in R for which there is a developer-defined transformation in T, and all of the classes in D. If CV is empty, then a scan of the store, phases ❺ and ❻, is unnecessary and is omitted.

❸ *Prepare for Evolution* — Carry out the cyclic garbage collector's marking phase for the whole store if CV is non-empty. Place a `starting_evolution` marker in the log as recovery behaves slightly differently for evolution.

❹ *Manipulate the Set of Persistent Classes* — Traverse the store's persistent class directory (PCD) and rename all of the old classes in R systematically, so that they can be used in the transformation methods, and can co-exist with their new forms. In the same scan remove them from the PCD so that they are not available for normal computations against this store. Insert into the PCD all of the new classes in R and N as they may be used in the transformation methods. Mark all of the classes in D so that they are not used to form any new instances.

❺ *Perform all of the Instance Substitutions* — Traverse the store one populated partition at a time in ascending order by physical partition number. For each partition, `p`, first perform a partition garbage collection marking phase, so that garbage objects are not resuscitated by evolution. Then for each class, `C`, in CV lookup `C` in `p`'s descriptor table.[1] If none are found, skip to the next partition. Conversely, if a descriptor for any `C` is found, scan the objects in `p`. For each non-garbage object that has a descriptor that refers to a class in CV do:

- **if** it has a transformation in T create a new instance in the format `C'` in a new partition, record the ⟨old-PID, new-PID⟩ pair in a structure called the `pidTable`, move data into it using the default transformation, and then apply any developer-supplied transformation.

[1]This is in the partition's header and hence only the start of the partition need be read. A trivial case is where `p` has a regime that could not hold instances, this can be determined without reading the partition header.

- **else** it has no transformation but needs one, so a violation of the consistency rules has occured and a roll-back of the evolution operation occurs.

The old and new worlds are now co-located into one new partition using the standard partition garbage collector to move objects between partitions. The old form of each instance is directly referenced by its PID but the *limbo* form lies hidden behind it, in the following bytes within the partition, no longer directly referenced by a PID.[2] The change in logical to physical mapping is recorded in the log, as is the allocation and de-allocation of partitions, as usual for a garbage collection. If the switch in the next phase does not occur, the limbo objects remain unreachable and are collected by the next garbage collection. Hence the reachable store currently hasn't changed, it has only expanded to hold unreachable limbo objects and new instances reachable only from them.

❻ *Switch from Old World to New World* — At the end of the previous phase all of the new format data and classes are in the store but unreachable. Up to this point recovery from failure, e.g., due to an unhandled `Exception` thrown by a developer-supplied transformation, would have rolled back the store to its original state. We now switch to a state where recovery will roll forward to complete the evolution, as, once we have started exposing the new-world we must expose all of it and hide the replaced parts of the old-world.

The contents of the `pidTable` are written to the log, so that there is enough information to roll forward. The set of partitions that can contain limbo objects or PIDs referring to them is discovered by scanning the log for records recording a change in the logical-to-physical partition number mapping. Each of these is visited and the old instance is converted to a dummy trivially collected at the next partition garbage collection. The limbo object has its header changed to a proper object and the entry in the indirectory is adjusted to point to this new object. The old descriptors are removed. All references to a new-PID found in the `pidTable` are replaced by the corresponding old-PID. Once this scan has completed, the new world has been merged with the unchanged parts of the old world.

❼ *Complete the Manipulation of Persistent Classes* — The classes that have fallen into disuse are uncoupled from data structures that might make them still reachable, e.g., in the PCD and in `ClassLoaders`' data structures.

❽ *Commit Evolution* — Release the locks taken to inhibit other use of this store and write an end-of-evolution record in the log.

The evolution technology is complex, but all aspects of it appear to be necessary. For example, the new versions of instances have to be allocated away from the instances they replace so that they can both be referenced in the developer-supplied transformation code. They then have to be moved, so that *only two extra partitions* are needed to complete the evolution, and no extra PIDs are needed in fully populated partitions. The garbage collector expands or contracts partitions as appropriate when it allocates the destination partition, but it cannot expand a full indirectory.

8.4 Summary and Directions

The implementation of the evolution technology lags behind the description above in two respects:

[2]This has the advantage that no extra PIDs are needed in the new partition and that extra space is only needed for the normally small proportion of instances that have evolved in each partition. But new data may reference this new instance via the PID at which it was created, hence we need the `pidTable` to fix up these references in phase ❻. There is a slight problem as developer-supplied transformation code may try to revisit these limbo instances while transforming some other instance, e.g., to form a forward chain. A solution to this, which complicates the Sphere interface, is being considered.

❶ If the instance format change were so large that it forced the new instance to be housed in a different regime, the evolution would fail.

❷ The verification is not yet as fully integrated with the tracking of classes that use a store as we would like it to be, nor does it fully support the build proccess, i.e., apply the Java compiler to all relevant classes in the right order and carry out compatibility checks over classes only available as byte codes (but considerable progress with these issues has been achieved [77]).

The first of these would require a new variant of the disk garbage collector, but the operation dispatching model of Sphere ($\leadsto$7) should permit this to be built easily. The second of these is currently being coded but the currently released command `opjc` does not carry out any consistency checks against the *unchanged* classes nor against classes that use the store but do not become persistent. We see this line of development as a mere step on the way to integrating development evolution with a developer's interactive development environment, build tools and configuration management system.

The current technology operates well against large stores and we have been able to demonstrate a wide-range of evolution steps. We know that some of our users use it routinely. As the development stores become larger and as the application complexity and longevity rises, we are sure that this use will increase to the point of dependence. Only field trials will tell us whether operations can be supported using this technology, until some form of deployed evolution is developed.

Research into improved mechanisms, e.g., a parallelised scan of the partitions in phases ❺ and ❻, will probably be rewarding and a useful precursor to evolving concurrently with the operational load,[3] Challenge 7 on Page 76. Research into the integration of evolution with other development tools is essential, Challenge 5 on Page 75, as is research into the best way of delivering deployed evolution, Challenge 6 on Page 76. By harnessing the information afforded by persistently maintaining the binding between classes and their instances, we believe there is a high chance of significant progress.

[3]This will be facilitated by the fine-grain isolation currently under development ($\leadsto$10).

Chapter 9

Platform Migration

Application developers want to take advantage of many technologies. For example, they wish to have their platforms run on the latest or most cost-effective hardware. They also wish to take advantage of other technical advances. In our case, the most crucial one is developments in the platform itself. These divide, in the current context, into Java platform developments and persistence implementation developments. Accordingly, we divide the discussion into: hardware migration, Java platform migration, persistence platform migration, and application migration tools.

9.1 Hardware Migration

Two forms of hardware independence need to be distinguished: dynamic and sequential. In the dynamic form, the stored data may be in a form for one architecture, and the executing virtual machine using that data may be on some other architecture. In the sequential form, the developer or application organisation may have worked with one hardware substrate and then need to run with another. Three strategies for handling this can be deployed, and may be combined:

❶ The data may be kept in some canonical format (e.g., XML) and a mapping to the format required by the currently executing VM is performed as each object is brought into the computation. This strategy is used by most database systems and is used in the archetypal enterprise service, SAP [167].

❷ When a VM first interacts with a store, the store's representation and the VM's format expectations are compared. If they do not correspond, the contents of the store are transformed to match the new expectations. This is the way many tools, such as office-productivity tools, deal with intercourse between users with different versions of the tool. It was also used in an object store that supported several persistent languages [36].

❸ The data can be kept in the predominant format, and most object-faults (transfers of an object from the store context to the VM) will not require transformation, other than pointer-swizzling. When a VM requiring a different format, e.g., one using the store over an NFS network, uses the objects, the format transformations are required.

The utility of these three techniques can be compared. The first always pays a computational cost and may in addition pay a space and, hence, bandwidth cost. It is therefore unattractive, even though it underpins many persistence mechanisms, such as mappings to relational databases, or Java Object Serialization ($\leadsto$13). Method 2 is only viable if the volume of the data is small and if there is only one VM allowed to access the store at once. In PJama we support multiple reader VMs or one writer VM and aspire to large stores, so this method is not suitable for PJama. Method 3 is the currently preferred method for PJama, with a comparison of formats occuring as the store is opened.

The commonest requirement for the format change is because the store is in big-endian format and the VM requires small-endian format (or vice versa). The necessary reversals are straightforward. However, as the Java programming language uses different sized fields from `byte` to `long`, it is necessary to know the field lengths. This can be handled by extending the role of the descriptors in Sphere.

A more difficult issue is when the size of values of the same base type changes. For example, the length of a reference field changes from four bytes to eight (or vice versa). This may be the reason for using the canonical store form (method 1) but it is possible to deal with this as the object is copied from the buffer to the object cache (or vice versa). Sphere's swizzling call-back can be used for this ($\rightsquigarrow$7.2).

Where sequential hardware migration is concerned, e.g., moving a store segment to a different disk, utilities are required, in order to maintain the Sphere data structures and to use locks to ensure that the store is not active while it is being moved. Where a format change is required, the stop-gap arrangement is to use the migration tool ($\rightsquigarrow$9.4) which operates off-line. To achieve endurance, both of these migrations have to be achieved incrementally as a background task. The former may be coupled with the disk garbage collection technology or with incremental archiving via the log ($\rightsquigarrow$7) and the latter with evolution ($\rightsquigarrow$8).

9.2 Java Platform Tracking

A recurrent task in the PJama project has been adapting the existing PJama platform to correspond with successive releases of the JDK, which define the Java platform standard. It is clearly necessary to carry out such tracking if the application developers and deployers are to see the advantages of Java platform improvements. That is, it is a prerequisite to enabling the users and developers to migrate to a new platform.

This has been an unexpected and draining consumer of effort ($\rightsquigarrow$A). Unless we devise improved strategies and architectures, Challenge 2 on Page 74, this demand would fall on a product team, if persistence became part of a standard platform. Reducing this cost is therefore a factor in demonstrating the OP hypothesis.

The platform changes fall into two parts: changes to the implementation of the virtual machine and changes to the set of core classes. An unsuccessful attempt to address the first part was conducted in conjunction with the subgroup of ODMG defining the mapping to the Java programming language in 1998. Here we sought to identify standard "hooks" that it would be reasonable to ask all JVM builders to support [34, 70, 19]. The endeavour was unsuccessful because of the diversity of views on what was required and the constraint that changes to the instruction set of the virtual machine could not be contemplated. Complications arise because the implementation of persistence uses core classes which are also used by applications.

This is a research effort that should be revisited as our understanding improves. It seems quite viable to achieve an extended instruction set with barriers [102], and to incorporate that into any specification of a *complete* virtual machine (barriers are necessary for other purposes, such as application containment and concurrent incremental main-memory garbage collection). We believe that eventually barriers will be afforded some hardware support.
The core class library changes are a serious work load at present, as is exemplified by Table 9.1.[1] The columns labelled with a JDK release number contain the number of classes in the core for that release and then the changes that occur between that release and the one to its left.

❶ The first row, *classes*, is the count of the total number of classes defined in that JDK. These include those in the public domain, i.e., named `java*` or `javax*` and proprietary classes with other prefixes.

[1]The data is incomplete, as we ran analyses only over the versions that we had on hand.

	Release of the JDK core classes						
Items Counted	1.0.2	1.1.1	1.1.5	1.1.6	1.1.7	1.2	1.2.2
classes	604		1622	1671	1719	4273	4337
removals	25		29	8	737	113	
additions	1043		78	56	3291	177	
changed	574		431	970	981	1256	
code only	180		126	888	457	695	
conservatively	296		74	74	396	467	
non-conservatively	98		231	8	128	94	
user impact	11		57	1	48	12	
reformat	173		229	60	266	338	

Table 9.1: Table showing changes between versions of the core classes

The growth in the number of classes is indicative of the added value of working with a commercially popular language. It also indicates that there may be continuous demands on a persistence team if new classes are being introduced which are not compliant with the resumable-programming style [120] (⇝5.6).

❷ The *removals* row indicates the number of classes that were removed from the core. They are typically not public interface classes, but part of their implementation. Nevertheless their demise can cause problems for application migration, as there may be persistent instances, through the reachability graph. There is a compliance constraint on the evolution step (⇝8.3 ❷) that it makes all of these instances unreachable in the new world.

❸ The *additions* row is the number of new classes introduced. New classes that are leaves in the class hierarchy should present no problems as there cannot be pre-existing instances of them. If, however, they are inserted as a super-class of some existing class, then consistency problems, and existing instance reformatting has to be addressed.

❹ The *changed* row indicates the number of classes that have had any kind of change made to them, fields added, removed or changed, or methods (including initializers and constructors) added, removed or changed. Compliance with new interfaces or new super-classes will appear here.

❺ The *code only* row is the count of the number of classes whose only change is to the code of one or more methods (with the usual generalization).

❻ The *conservatively* row shows the count of the number of classes whose changes could not result in any existing class that uses them being unable to function [76], e.g., they added members.

❼ The *non-conservatively* row is the count of classes which, through the removal or redefinition of their members, no longer offer the same interface. Although the majority of these are not in the public domain (see ❷ above), an application developer is not prevented from using them. If this has happened, then the consistency check on a populated store would show that migration to the next platform was impossible without first changing user classes.

❽ The *user impact* row shows the count of the classes that are non-conservatively changed and that are also in the public domain. These present the problem just described, but with higher probability that it will occur.

❾ The *reformat* row is the count of the number of classes whose changes must result in a change to the format of all of their instances. If there is a population of these instances an evolution step must be performed. However, typically this is impossible because there is no transformation specified and a default transformation will not suffice. Only when we have experience with deployed applications will we be able to estimate how many of these actually have instances in stores and therefore present a problem.

The final comment shows a significant impact of considering persistence. If developers or users are to trust any persistent platform, they must be assured that platform migration is possible. Otherwise they lose their data at the point when their existing platform technology is inadequate. But for them to receive that assurance, those who provide class libraries must provide evolution transformations corresponding to releases. This takes significant time and resources which language developers do not factor into their planning and costing. In contrast, reliable database platform vendors, CAD system vendors, office-productivity tool vendors, etc., already recognise the need to deal with this aspect of migration and ship their products with version-migration code included. It is not yet apparent into which of these camps EJB component providers fall ($\rightsquigarrow$13).

9.3 Persistent Platform Migration

An advantage of a small API ($\rightsquigarrow$5) is that very little of the application code is affected by changes in it. Indeed virtually all of the effects of orthogonal persistence take place implicitly and automatically and new implementations of these have supported old PJama applications without any changes. We have revised our API, for example simplifying the interface to the persistent roots by taking advantage of a newly available collection type. These changes have been so small it has been straightforward to emulate the old API for a changeover period.

Whilst programs have ported straightforwardly to new versions of PJama they have not been able to continue using the same stores. The reasons for this are that the platform is implemented with many internal C and disk data structures and system classes that change between releases. These data structures could not be translated automatically for two reasons:

❶ They are not described in a way that would guide transformation.

❷ They are needed intact in the old form to scan the existing store and are themselves below the depths of store introspection and their new versions have to be present before a new store can be populated.

This problem is overcome by a logical archiving system described next.

9.4 Migration Tools

The logical archiving tool `opjlarchive` [172] scans any populated store and makes a copy of it in a canonical form that is independent of the hardware platform, of the implementation of core classes and of the implementation of PJama including classes defined as part of that implementation. The complementary tool, `opjlrestore`, can be used to reload that data into any version of PJama.

The principle of this tool is to scan the total reachable graph using reflection, marshalling an image of each object to an output file in the canonical form when it is first reached. As each object is encountered a map is built from the object's identity to a dense range of the Naturals. These Naturals are used as a surrogate for object identity in the canonical form so that any object graph can be represented independently of identity allocation policies. The map also provides a means of avoiding rescanning common substructures and cycles. All scalar fields are marshalled to forms which are not sensitive to the representation of scalars, e.g., endian-ness. Any class that was loaded by an application loader is marshalled into the canonical stream in its byte-code form, and classes loaded by multiple class loaders are referenced from the data structure representing that loader's loaded classes. Properties of objects, such as: their identity hash code, the variables which are marked transient and their registered listeners, are all properly preserved.

The archive output from that scan is sorted using a topological sort so that every class will be loaded before any of its instances. This can be done at any time and does not require access to the store, which can therefore be in use. The restoration scans the file of canonical data and reconstructs each object in turn, and a new version of the map from Naturals to objects. At the end of reconstruction this map is used to fill in any forward references (there will be some even after a topological sort if there are cycles in the graph). Bindings or references, that were previously to system or platform implementation classes, are fixed to refer to their new equivalents. Finally the new data structure is checkpointed into the new store.

The reflection facilities had to be extended to make restoration possible. For example, instances had to be created without running a constructor and then the values had to be written into the instance. New facilities were needed to write in its identity hash code, for example. Similarly classes had to be installed and resolved without running their initializer and then their transient markers, identity hash code and static variables, had to be set.

9.5 Migration Conclusion

We have in logical archive and restore an adequate mechanism for initial experiments. It has yet to be tested with a wide range of stores but we believe that it should work. Moving between platforms is a relatively rare event and it is usually an expensive process. It will eventually become necessary to build a concurrent logical archiving mechanism that can be run alongside the application load. This is clearly necessary if high availability is to be achieved, as the new platform would need to be prepared ready for a hot swap. This would be driven by combining the scan with updates derived from the log, as with incremental physical archiving. The fine-grained isolation under development should facilitate this ($\leadsto$10). The installation on a new platform by a logical restore would still be in stand alone mode. However, it must become incremental, to cope with envisaged scales, and it may be parallelised to reduce the update from the log at the start of the change over of service platform.

Chapter 10

Transactions

The word “transaction” has come to mean, in many people’s minds, the ACID transactions of OLTP [92]. Transactions were introduced into database programming for two reasons.

➊ To delimit units of work, so that they could be controlled and managed by a DBMS.

➋ To provide isolation so that application developers could write code with the illusion that only one instance of their program would execute at any one time and no other programs would be running.

The former permits the DBMS to forcibly rollback a transaction that is causing a deadlock or that transgresses some rule, and it allows it to choose which transactions to run and when to run them. The latter permits programmers to avoid having to think about synchronising with all other programs about race conditions, resource hogging, etc. It can be contrasted starkly with the conditions that prevailed prior to the advent of transactions, e.g., with early CODASYL databases. Then programmers were required to request and release locks explicitly in an attempt to make programs work concurrently. Few application programmers ever mastered the art of using locks in this way and often degenerated into massive overlocking. The conceptual problems become insurmountable as more and more people work on each program, as the database progressively evolves and grows more complex, and as the set of programs increases to meet the growing complexity of enterprise applications. ACID transactions solve that problem.

10.1 Improving Implicit Locking

The same case for delimiting units of work and for supporting programmers with isolation can be made in the context of a single Java virtual machine [54]. Here, as the enterprise applications take on more challenging tasks, they use more threads, they are written by more programmers who use large numbers of classes supplied by other organisations. One of the strengths of the Java programming language is that it encourages programmers to write multi-threaded code, and another is that it is easy to compose classes from many sources. So we see the same confusing programming context emerging again. What can programmers do? They probably once again adopt a defensive position and over lock, typically by the excessive use of **synchronized**. But this leads to frequent deadlocks or poor performance. A symptom that there are these problems is recent changes to the core classes that remove synchronization there. This passes the problem to the developers of applications without resolving it. Without a smaller delimited unit the JVM can only resolve the deadlock by terminating the whole application, which is hardly compatible with service availability and reliability. The EJB specification addresses this problem by disallowing concurrent access to bean instances, and prohibiting both the use of **synchronized** and the creation of threads by bean instances.

This somewhat Draconian position is mitigated by the prospect that the EJB server will manage concurrency control automatically.

The DBMSs have well developed automatic locking mechanisms that permit them to provide isolation via implicit locking. Locks are claimed and released automatically, with the DBMS choosing the granularity, imposing a protocol and managing internal structures specially to gain higher levels of concurrency. Their approach does not translate to the JVM context as it is too heavyweight.

DBMSs can afford to use locks of relatively high cost because the interaction between programs is inevitably at a relatively low rate. All of the interaction with the DBMS, the only place where these programs might interfere, is via inter-process communication, often via several steps through processes and even between machines. The round trip cost and delay is so high that database interaction has evolved to use large units of transfer between program and database and to minimise the interaction frequency. The earliest examples of adaption to do more per database interactions were relational queries returning bulk results or performing bulk updates. The trend now is to use database procedures, so that their parameters are passed, control is handed to the DBMS, a complex batch of operations is executed, and then a collection of results is returned to the program. This is exemplified by the `execute` method of `java.jdbc.Statement` and by the use of static methods as database procedures in SQLJ [46]. The individual activations of these database procedures do not share state, except via the database.

In contrast, within a program, threads interact at a very fine grain, reading or writing instance variables in a few instruction cycles. Sustained research by Laurent Daynès [68, 69, 66, 67, 58, 63] has recently led to major progress in reducing dramatically both the time and space overheads of implicit lock management in object-oriented virtual machines. The extent of these achievements and the implementation techniques used can be found in [59, 65].

This is one step towards realising the goal of a flexible transaction mechanism proposed for PJama [16, 20]. Similarly, Sphere has been equipped with multiple histories mapped onto one physical log, so that concurrent but isolated threads of execution can separately record their state changes and roll back or checkpoint independently ($\rightsquigarrow$7.3). But this is not the full story of managing flexible transactions, as we will see after a digression to consider another approach to isolation.

10.2 Complete Isolation

If all that is required is to isolate whole applications written in the Java programming language from one another, then several methods are possible. The simplest is to use operating system facilities, create separate processes and run separate instances of a JVM in each process and run each application in its own JVM. This has the advantage of depending on trusted and hardware-supported isolation technology, but the disadvantage of incurring high costs in memory usage, start up processing and inter-application communication [54].

An alternative is to rely on the strict enforcement of the type rules and security manager in the Java platform [91]. If this is made to work, it has the advantage that the JVM at least is shared. This kind of isolation has been necessary to sandbox applets and servlets. In those cases, it is based on using a different class loader for each isolation region, as the Java platform treats the same class loaded by a different class loader as a different type and precludes their interaction. When this approach is used isolation is not perfect. For example:

❶ *Service Denial* — an application may keep constructing objects and chaining them from an active thread until it is hogging all of the heap and denying other programs service.

❷ *Interaction via Core Classes* — applications may interfere via their use of core classes (static variables and synchronization monitors) that haven't been replicated.

❸ *Wasted Heap Space* — each time a class loader loads a class it allocates the space again for its byte codes which then get expanded by JIT compilation for each copy of each class.[1]

An experimental system by Grzegorz Czajkowski [54] has shown that the code of classes can remain shared but applications can be completely separated by replicating class static variables and carefully chosen synchronization monitors. This overcomes problems ❷ and ❸ and gives complete isolation apart from the risk of service denial (❶) which it could be extended to overcome. At present it works with the standard KVM [203], but it could be applied to PJama$_{pt}$ and with additions to deal with such things as native methods and threads to PJama$_{1.0}$. Such a system only provides isolation; any controlled sharing has to be implemented outside of the applications, e.g., via a “database” or by significant changes to the model.

10.3 Isolation with Controlled Sharing

We perceive a trend in modern applications towards greater interworking between applications. A few examples are:

❶ When data about a person is manipulated or the application is being used to communicate with others, it needs to interwork with the address book application.

❷ When a fault occurs either in the application code or the processes it is helping to support, it needs to report this via the mailer application.

❸ When the application handles numeric data it needs to interwork with spreadsheet, statistical analysis and graph drawing applications. These may then need to interwork with presentation applications, visualisation tools and computer-supported co-operative working (CSCW) tools.

❹ When an application handles time, it may need to interwork with calendar and scheduling tools.

❺ When an application is used to build digital inventory, such as source code or web-site material, it will need to interwork with a version and configuration management system.

This list of interworking requirements is growing rapidly as the expanding population of geographically distributed professional workers and of personal application users gains experience. A consequence of this experience is that once they have seen integration work well in one context, say an office productivity suite or a personal organiser, they expect it to scale up and work everywhere. They will no longer tolerate error-prone techniques such as running one application, extracting data from it, and then pasting it into another. This is particularly irritating if format changes are required, or this human implementation of interworking is required repeatedly [109].

10.3.1 Conventional Support for Interworking

Application developers use one of three responses to meet this requirement:

❶ *Bundling everything* — The developer meets every additional requirement by bundling into the one application the code to do everything, importing applications as packages or writing limited versions of the applications. Componentisation systems, such as Enterprize Java Beans^{TM}, assist with this.

[1]This is an implementation weakness that could be fixed.

❷ *Using a standard inter-process communication protocol* — Each application runs in its own virtual machine and process continuously as a service. Interworking is achieved via interprocess communication: Java RMI, CORBA, DCOM or XML exchanges.

❸ *Communicating via databases* — Again, each application runs in its own virtual machine and process, and communicates with other applications by retrieving information they have written to databases or posting data in the database for other applications to read. The database is the locus of sharing. It supports data sharing directly, and via trigger actions, database procedures and signalling through the state of the data. It supports interworking indirectly and inconveniently.

In reality, many modern application suites will use a combination of all three of these, but it is convenient to separate them in order to review their strengths and weaknesses.

The Re-emergence of the Complex Locking Nightmare

Approach ❶ has the advantage and disadvantage that everything is intimately interconnected. All of the objects are in the same address space and they can directly use any of the variables and methods that other objects expose. This can result in rapid responses to interactions between the components and hence rapid responses to users. The problems are that the developer is faced with the complexity we defined above. In this context deadlocks are frequent and there is nothing a controller can do to unfreeze them safely. Furthermore an error in any of the imported components will cause the whole combined application to fail. Because the components are independently developed, it is difficult to arrange to checkpoint them all consistently. Programming conventions are introduced to try to overcome these problems, but they are not enforced and are difficult to follow precisely. In consequence, this approach cannot be considered safe. Finally, the workers who are end users of these applications are often multi-tasking, using several applications at once. If the different suppliers of these applications have chosen different sub-application components, these multi-tasking end users will be frustrated by inconsistent interfaces. Furthermore, they probably have preferences as to which version of a sub-application they use, but this will be over-ridden by the developers' choices.

Dependency on Process-based Protection

Approach ❷ offers improved safety and consistency compared with ❶. As the applications run in different processes they cannot directly interfere with each other's state and a failure will normally be localised. They can still get into states where there is a distributed deadlock, they may have even more difficulty co-ordinating a checkpoint, and they may be susceptible to denial of service by one sub-application. Though operating systems have advanced and reliable mechanisms for isolating and monitoring processes, they cannot tell whether inter-process communication is appropriate — they've been known to support message live lock for days. A drawback of these systems is that the cost of intercommunicating is high. The marshalling and unmarshalling of messages, together with communication protocol overheads and process switches, all cause delays, incur CPU overheads and generate space demands. Spence has noted that this is exacerbated when persistence is well supported, as larger structures are available and are often copied by the default protocols [195]. This leaves application developers with the additional challenge of modifying marshalling or partitioning structures. Finally, this approach has to run many copies of the JVM and other infrastructure, and it either has to pay the cost of launching services on demand or of running them continuously. With agreed standards, the users can see consistent applications, as they will interact with their chosen version of each standard.

The Database as Mediator

Approach ❸ has the isolation strength of ❷ and in addition it overcomes the problem of deadlock, livelock or misformed communication between services, since integrity constraints, database procedures and the control mechanisms of DBMSs are designed to limit these problems. Co-ordinated checkpointing becomes a problem when there are multiple databases involved and a suitable protocol is required. This has all of the overheads of ❷ and additional overheads due to the need to transform representations to and from database formats and to pass all messages through a succession of DBMS processes and via a database as an intermediary. This intermediary means that sub-applications are not notified of significant changes and cannot be invoked easily to carry out a specific task on behalf of another application. The resource costs now have the added cost of running the DBMS processes and traffic between them, though some of these may be shared among many applications. Typically, each interaction between an application and a database is cast as an ACID transaction. This makes it possible to guarantee integrity and durability in each database and move them between self-consistent states. The developer gets no help with consistency within applications and between databases. Essentially, isolation and database reliability have been achieved at the cost of restricted and slow interworking between applications and considerable development complexity.

Having our Cake and Eating it

The question arises, "Can we have the safety of ❷ and ❸ but the efficiency of ❶?" We believe that this is possible.

10.3.2 Controlled Interworking

The challenge is to enable applications to safely and intimately co-exist within one virtual machine, so that their interactions can be rapid and fine-grained, and yet to inhibit all unintended interference. Databases have shown that implicit and automatic locking can isolate applications. The high-performance implicit locking mechanism for objects ($\leadsto$10.1) makes it feasible to replicate that model of isolation and control within a JVM. The associated separate virtual log, *history*, provided by Sphere makes it possible to forcibly or voluntarily roll back individual, isolated sub-applications to their earlier checkpoints.

The properties of strong typing, such as found in the Java programming language can be exploited to conduct data-flow, control-flow and escape analyses to reduce the cost of this isolation further. We have yet to exploit this.

To admit greater communication between applications (or sub-application components) it is necessary to relax the isolation enforcement in a controlled way. There are many examples in the literature of specific relaxations, yielding new transactional models [80]. In each case, they choose a specific balance between isolation and interworking and impose a particular temporal model. What is required is a general-purpose mechanism that permits these and others to be formulated and enforced in this new programming context [64]. There are two components to this problem.

❶ An implementation mechanism is needed which is flexible, robust and efficient.

❷ A notation is needed to specify exactly what is permitted and what is denied between pairs of applications.

The originally proposed flexible transaction model for PJama, [16] suggested using a conflict resolution technology (where the implementer of a transaction framework could indicate that certain lock conflicts should be ignored), and a delegation mechanism that allows a transaction to pass responsibility for a set of objects and their locks from one transaction to another. This has been explored and is currently being combined with the new locking technology, FIT, and transferred to PJama$_{1.0}$ [65, 60, 61]. It should soon be possible to take applications written to run in separate JVMs and to run them within a single JVM safely under the control of this transaction framework *without making any changes to the applications*.

Transactions and Structural Definitions

A major challenge that remains is to provide a succinct and tractable notation for describing exactly how applications are permitted to interact, since manually implementing adjudicators for each category of collision violation would be laborious and error prone. Apotram [8, 9] is a candidate notation and the recent work cited above has shown that it can be mapped to and supported by FIT.

It seems likely that application developers may benefit from using the facilities of the Java programming language that allow interfaces to be specified in conjunction with some scheme for structuring the application into independently maintainable zones [218]. Then the isolation system could enforce the zone structure, ensuring that only intended interfaces were used in pairwise agreements to restrict information flow and dependence. We hypothesise that it will be profitable to integrate such application structuring designed to aid software engineering processes with the run-time isolation and sharing management. This will make the software production processes more reliable and may allow significant reduction of run-time costs via static analysis whilst delivering the required safety and control.

The development of the potential of this line of research remains an identified Challenge 12 on Page 77 which we will continue to pursue.

Chapter 11

Distribution

Two general approaches to distribution are:

❶ A federated model, where data and code remains in its location, the distribution is visible and processes interact via messages, such as RPC.

❷ The distributed shared memory model (DSM) where the distribution is hidden and data, code and threads move about implicitly.

One way to combine persistence with distribution is to treat remote objects exactly as we treat objects on disk, that is, if there is a remote reference and the program tries to dereference it, the object is automatically faulted into the running JVM, effectively providing an object DSM. Although variations of that approach have been investigated [215, 165, 152, 154, 153] there are several arguments that prevent its simple adoption.

❶ The remote object may depend on its context, for example, it may display to a local screen or read from a local device, and hence it may be logically incorrect to implicitly change its locality.

❷ Object identity is compromised as objects are replicated, possibly via a cyclic chain of PJVMs. Expensive mechanisms are necessary to re-instate identity.

❸ If remote objects are copied one of two problems ensues. Either the semantics of an object are changed (because there are two copies, updates to one aren't recorded in the other), or an expensive cache-coherence protocol must be supported.

❹ If a remote object were copied implicitly, then it might have many more references to other objects; if the transitive closure of these references is copied, this may be a very large graph (particularly in a persistent context when it may fill many disks and be accumulated over years), so either an expensive copy is incurred or there may be a cascade of remote fetches.

❺ Once stores hold references to other stores that can be dereferenced at any time, the referenced stores are subject to an indefinite obligation to hold the referend and its dependent graph. As the stores holding the reference may never be activated again, this could be a serious cost. In any case, it may ultimately require a distributed cycle removing garbage collector to untie the knot. Once again, persistence allows these graphs to grow larger and more complex.

❻ Errors are more common over networks, particularly WANs, than they are with disk technology. In consequence there are additional common failure modes, recovery from which is best achieved with application specific responses.

These boil down to two basic reasons for not fully automating remote access along the lines of the persistence principle: the stores become entangled with a serious rise in operational management and a loss in autonomy, and the application developer has to make choices and program responses for distribution — we have not yet found a general solution. The former effect is exacerbated by the longevity of persistent systems.

11.1 From Java RMI to PJRMI

We therefore chose to seek a compromise, where we aim to help the application developer combine distribution with orthogonal persistence. The standard Java RMI facilities were augmented with persistence support, which we call PJRMI [194]. In Java RMI a developer designates whether an object is to be copied or whether a reference to it is to be returned, in which case a stub is created so that the object is used automatically via remote method calls. This depends on the standard Java RMI technology; objects that are instances of a class that **implements** `java.rmi.Remote` are represented via a stub and all others are copied, including their reachable graph until either a `java.rmi.Remote` object is reached or the default marshalling has been overridden. So far this is the standard Java RMI mechanism. The extensions provided by PJRMI, which will interwork with standard JVMs so that the same code works for all combinations of persistent and non-persistent communication, do two things.

❶ They provide resumption semantics for remote references. This is implemented by storing the PJRMI stubs in the persistent object store and using the `OPRuntimeEvent` system (⇝5.5 Page 22) to restore them on restart. In fact, it restores an incremental mechanism that resumes the individual socket connections and stubs on demand.

❷ It implements a session and lease system that limits the liability of the stores that have exported a remote reference [196, 195]. Communication with a remote store takes place within a session, which the store must have agreed to. Then leases are granted with an elapsed time, long compared with clock synchronisation, e.g., days or weeks, after which the issued stub will raise an exception rather than attempt to resume communication. The lease-granting store "knows" that if a lease has expired and there are no other references, then a data structure can be garbage collected.

We do not consider the present mechanism to be a final position. For example, the distributed garbage collection used by Java RMI could be extended to help reduce inter-store commitments more rapidly, and a lease renewal scheme could be introduced.

11.2 Beyond adding Persistence to Java RMI

Our major concerns, however, are with performance and with evolution. The algorithms inherited from Java RMI use Java Object Serialization (JOS) to marshall the transitive closure of reachable objects, with the boundary conditions given above. This graph may be very large and the cost of marshalling, unmarshalling and storing the copy may be high.

Susan Spence has therefore carried out experiments with various heuristics that copy this graph of objects in incremental batches. These have suggested that this will pay off, but the technology has not yet been incorporated in the released version of PJama. These experiments and details of the implementation of PJRMI may be found in [195].

The standard Java RMI provides some support for evolution. The default behaviour is to demand that class definitions are absolutely identical at each end of the communication. This can be over-ridden by asserting classes are the same, by setting a value in the **static final** `SERIAL_VERSION_UID` the same for the similarly named class at each site. In fact, programmers have to do this to mix persistent and non-persistent JVMs. This mechanism allows a limited set of changes, basically addition of fields that are then ignored. It would not cope with the changes we are able to achieve with PJama's evolution technology ($\rightsquigarrow$8).

11.3 Summary

We have at present a pragmatic and workable solution to allowing developers to use the standard Java RMI in conjunction with persistence. There is still potential for further development of this facility, but the main challenge is to introduce heuristically guided incremental algorithms to allow scaling and to find ways of combining evolution with established communication commitments, Challenge 13 on Page 77.

Chapter 12

Experience and Performance

A motivation for developing PJama was to evaluate the utility of orthogonal persistence ($\rightsquigarrow$2). This section reviews progress with initiating that evaluation; the experience with using PJama for applications is reported and then we review some of the measurements of its performance.

12.1 Experience with Applications

Approximately 160 sites have taken out a licence to use PJama for evaluation and there are favourable reports in the literature from several of our evaluators [183, 27, 38, 39, 83, 114, 85]. There are several experiments with PJama within Sun Labs, including a novel web server, built by Stephen Uhler, and use of PJama$_{1.0}$ to support conceptual modelling [226, 7, 227], but none of these have reported results. The conceptual modelling system was converted to use the PJama$_{1.0}$ platform in about half a day.

To approach a reasonable standard for evaluating PJama requires that it be used by independent workers, that the tasks undertaken are in some way representative, and that the work is organised as an experiment, or at least there is some formal evaluation of the experience [209]. We have only three examples that begin to meet these requirements on an evaluation.

❶ The first evaluation was organised by Dag Sjøberg of Oslo university [93]. His student measured the use of facilities supplied by various Java class libraries for a set of Java applications intended to solve the same problems. For example, one package was `org.opj` and another was `java.io`. The set of programs were constructed mainly by students as part of exercises. In consequence, their size and quality was not representative. An overall simplification was shown by each of the measures when PJama was used, but the study was too small scale, used programmers of too little experience, and was too short-term, to provide conclusive evidence.

❷ The second evaluation [195] was conducted by a member of our own team, Susan Spence, and therefore fails to be independent. It was also investigating a particular aspect of PJama the provision and use of persistence enhancements to Java RMI ($\rightsquigarrow$11). In this case the programmers observed were experienced and independent researchers, in France [38, 39] and in Australia. The observations conclude that these groups were using `PJRMI` successfully and that they continued to use it over several years of a research project. By implication, since they can't use `PJRMI` without using PJama, they found PJama a viable platform.

❸ The third evaluation was conducted at St Andrews University by Graham Kirby, who used PJama as he built a reflection-based general purpose browser [160, 126, 128, 129, 127, 230]. This research was particularly focused on reflection and only used early versions of PJama; however, it was able to provide browsing over the contents of persistent object stores. It suffered from the problem that, particularly with earlier versions of PJama, there are difficulties handling `java.awt` classes in the context of persistence (⇝4.2.1).

One other independent group has regularly reported their experiences to us (often as bug reports!). This group is building web indexing and information retrieval tools [85]. They extol the convenience of orthogonal persistence, and have found the newly delivered scalability of PJama$_{1.0}$ a significant advantage, as they can now index the bible in one run, whereas previous versions of PJama had required nursing through the task in small increments. They have now indexed large batches of web pages, but have hit further scaling bugs.

Although not valid as a usability assessment, our own experience is a useful indicator of potential. Several projects have been undertaken, a sample of which are described below.

❶ *Geographic Applications of Persistence*: Over a period of three years, a succession of students at Glasgow have built a geographic information system running on PJama [113]. It will operate with all of the UK Ordnance Survey Strategi data [210], equivalent to all of the 1:50000 scale maps, or with collections of data from the US Bureau of Census Tiger data set [211], e.g., the whole of California. The features it supports include: user-control of which geographic feature types displayed and how they are displayed, a nonary tree index to support rapid panning and zooming, a name index to find features rapidly, and user-profile facilities for users to add their own data and to remember their viewing preferences. It is a successful demonstration of the power of orthogonal persistence, as this functionality could not have been achieved by the students if they had also had to map data structures to and from their persistent format (the Californian data is 3.2 GBytes which would not be easily handled by other persistence mechanisms for the Java platform — PJama allows the code to randomly access that data even though it is larger than the JVM's address space). Care had to be taken to keep certain `java.awt` classes and `Thread` instances from becoming persistent (⇝4.2.1).

❷ *Persistent Swing Demonstration*: The original Swing Set classes demonstration included with the JDK took a considerable time to start and lost its state on termination. Bernd Mathiske confronted the challenge of making resumable enough of the `java.awt` classes and the few `javax.swing` classes that needed special treatment (⇝4.2.1). This led to a version of the Swing demonstration application that restarted very quickly, because of PJama's incremental faulting policy, and which resumed exactly as it had appeared when last checkpointed. Here we see both the value of orthogonality and of incremental faulting algorithms on restart.

❸ *JP Configuration and Version Management*: This was our original target application [122, 123, 213]. The multi-threaded server that managed a version and configuration repository in a persistent object store was made operational quite quickly. This then needed remote clients to support collaborating engineers changing their source code, rebuilding derivatives and manipulating their version indexes. An initial version was built using `PJRMI` (⇝11). It ran into two difficulties: sluggish performance because the level at which operations were partitioned was inappropriate and generated too many remote invocations, and there was no resource available to build the interactive development environment that would have made JP usable. Once again, the platform supported the workload, resumption mechanisms restarted the connections and the server-side software ported easily. But the same care had to be taken to avoid the exceptions to orthogonality and to avoid the scaling limitations inherent in early releases of PJama. The JP infrastructure is still used in the JPCM benchmark, which simulates multiple users performing configuration management and editing tasks.

❹ *News Server*: With a view to stressing PJama's ability to run continuously, Neal Gafter built an NNTP server. This acquires the news group broadcasts and caches them in a persistent object store. It has recently started running on $\text{PJama}_{1.0}$ but has not yet run continuously because of an intermittent bug. From the coding point of view, this application demonstrated a deficiency in the API ($\rightsquigarrow$5) due to the absence of persistent threads. Currently, we record neither threads nor the lock state of objects in the persistent store. In a multi-threaded program, one thread may perform a checkpoint while other threads have not yet restored the invariants of data structures that they are modifying. The resulting persistent store contains inconsistent data. To address this problem we added an interface that applications can use to declare regions of execution within which the PJVM should not checkpoint. Unfortunately, this requires that applications be modified to use this interface consistently, which would not be required in the absence of persistence — a violoation of persistence independence. Nor will it be required when we achieve orthogonality completely.

❺ *Prototype Zoned Evolvable Distributed System*: Huw Evans and Peter Dickman developed the second and third versions of their distributed software architecture, DRASTIC, on the PJama platform [82] and an evaluation of their experience is given in [83]. This mainly concerns distribution and persistence. The very significant benefit of orthogonal persistence over Java Object Serialization is reported in [81], a view endorsed by David Jordan [115].

❻ *Web Server*: At Glasgow we have been building up a bioinformatics service and have needed a web server and a way of running Java Servlets so that they could access PJama stores. Initially we tried converting JigSaw [214] but found that its whole structure was shaped by the assumption of persistence based on files. We then ran a separate server and an adjacent PJVM on the same machine and communicated with that PJVM from the applets. This introduced extra communication delays, and in any case we are interested in what is a good architecture for web servers given an orthogonally persistent platform.

Huw Evans has therefore built a web server running on $\text{PJama}_{1.0}$. It caches web pages and holds all of the configuration and management data in the persistent object store. This is in preparation for delivering the service described in ❼. We expect to hold front-line data in the same store as we use for the web server, so that we can deliver a fast initial response to client applets, but to refer major tasks to other PJVMs due to the scaling limitation (32-bit addressing) and lack of fine grain protection ($\rightsquigarrow$10.1 and $\rightsquigarrow$10.3.2). We plan to exploit protection within the address space of the running PJVM as soon as it is available to allow user code to be uploaded to derive data from very large stores.

❼ *Services for Collaborating Geneticists*: Over the past two years we have built a suite of applications that are intended to grow into an infrastructure to support distributed communities of biologists using genetic and other data [105, 106, 108, 109, 110, 107]. We began by supporting a number of genetic-map viewing applets, so that biologists could view the various features that had been identified in a DNA sequence. The next step was to allow them to upload their own data or edits, which we store in the persistent object store for them, as time-stamped versions. This differentiates our work from the typical gene map viewing site, as it supports the iterative process of a team trying to make sense of a chromosome once they have parts of its sequence.

We are currently extending our work to support searches over the available DNA, RNA and protein sequences. There were about 600 megabases (Mb) already publically available. Recently, the whole human genome (approximately 3Gb) has been released. Our biologists say that they would like to search for sequence matches over all available sequences. We have been exploiting the properties of PJama to build complete suffix-tree indexes over DNA sequences. For example, the smallest human chromosome, 22 — recently sequenced [79], is 34Mb. A *standard* Java application building a suffix tree, exhausted a 2Gbyte heap, the current address limit, after indexing 27Mb. Improvements in coding might make that complete within the 32-bit address space, but this is beyond hope for the largest chromosome, about 340Mb. Orthogonal persistence allows the tree to be incrementally constructed and retrieved incrementally; even though it is much larger than the address space, only a small fragment is visited during a search. Of course, such a mechanism could be hand-crafted to store the data on disk, but the cost of maintenance in a field changing as rapidly as bioinformatics might then be punitive.

There are many other small applications that have been built by students, including a distributed bibliography manager that is intended to help a group of researchers maintain a shared bibliography as well as their personal views, private annotations and additions. Perhaps of more interest are reusable components that have been built, and which combine well with PJama, these include:

❶ *English Information Retrieval Tools*: A set of classes for carrying out the Porter algorithm for stemming and for doing an inverse weighted retrieval from collections of `java.String` instances is operational.

❷ *DataGuides over Java Object Graphs*: The DataGuide technology was originally intended for optimisable queries over semi-structured textual data [90]. Peter Morton has implemented a DataGuide that uses reflection to work over any Java object structures [162].

❸ *Visualisation of Properties of Large Populations*: Since the original starfield viewer technology was demonstrated as a means of enabling untrained users to pose queries and comprehend large volumes of data [3] we have explored its use with persistent data. We now have a Java class library that allows viewing of heap, persistent or file data with extremely good control over the viewing, filtering and mapping parameters, inspired by data mining tools [53].

We now have enough experience to confidently conclude that programmers find it easy to write applications using PJama. It is much harder to take an application that has used some other model of persistence and transform it to run on the PJama$_{1.0}$ platform. We had previously underestimated the impact on program structure that other persistence models have. On the other hand, we have never encountered any difficulty in porting to exploit orthogonal persistence a program that had been developed simply to run in main memory. We still encounter difficulties with the scale limits, orthogonality limits (⇝4.2.1), and bugs due to the limited resources we have to invest in perfecting the platform. But it is already a work horse, and the limitations are rapidly being overcome.

12.2 Initial Performance Measurements

A report on our experiments to measure aspects of the performance of persistence provision for the Java platform and the current results that they deliver, is in preparation [117]. It permits us to make provisional comparisons of the performance of different approaches and implementations. First we discuss what should be measured and introduce the background to interpreting such measurements.

Any persistence mechanism adds an increment to the overall cost of an application, but the size and nature of the increment varies markedly with the mechanism. The traditional programming style of serial input/output that brackets a period of computation has high costs at the start and end, but the computation phase runs as fast as possible. Applications that frequently access external databases using, say, JDBC, have a smoother performance profile, but can still endure lengthy bursts of input/output when crossing the process boundary to the database (⇝10.1). In contrast, orthogonally persistent systems typically have a much smoother performance profile, but incur a more or less constant overhead. It is an inevitable consequence of orthogonality that access to almost all objects must be mediated by the PJVM, for example, to perform a residency check, and this has an associated cost.

In the measurements that we have made on standard benchmarks, we see the OP overhead at 15% on average [136]. It is probable that this could be very much reduced by elimination of redundant barrier checks. However, comparing with purely transient programs is not really a sensible metric, as the storage and retrieval of long-lived data is intrinsic to large-scale and long-lived applications. Here though, comparison is much less straightforward. First there is a logical problem that many of the applications that exist today have been shaped by assumptions about what is feasible with existing persistence bindings to file stores, relational databases, and in a few cases, object-oriented databases. Second, there is a serious practical problem. Programmers are trained to think in terms of conventional mappings of application tasks onto traditional persistence technology and their programs are completely shaped by that technology and the training. Consequently, it is very hard to transform a program from that (we would say distorted) form into one which simply follows the application's "business" logic. But as we have remarked already, interesting measurements are those from real applications, not from artificial, micro-benchmarks. Third, all the measurement attempts conducted so far have shown enormously varied results, including speed-ups using PJama$_{1.0}$ by factors of 1000, that are very difficult to interpret and report. Fourth, many stores deteriorate as objects are moved, usage patterns change, etc. Few studies, if any, have investigated these longer-term phenomena.

With these caveats in mind, we turn to a discussion of the dominant factors in the performance of an orthogonally persistent platform. Since we strive to support continuous computation, overall throughput is the natural measure, and this breaks down mainly into the following three aspects.

- ❶ The time it takes to resume an application to the point where it can operate effectively.
- ❷ The effect of the extra run-time costs on the performance of the application once it has faulted in its working set of objects.
- ❸ The latency of the checkpoint operation, where modified objects are written atomically to stable storage.

We will consider these in turn.

12.2.1 Resumption Performance

The perceived need for fast resumption is highly application dependent. In a world where applications, structured as individual operating-system processes, come and go frequently, the perceived need is high, especially if the application has an interactive component. Indeed, recent development of the Java platform has focused on shortening JVM startup time. On the server side, however, it is expected that applications are much longer-lived and therefore resume only in the event of a crash or an infrequent explicit shutdown. In either case, OPJ provides an inherent benefit since, after the first run, the computation is held in its suspended state in the the stable store and can be resumed quickly by incrementally faulting-in just the needed objects. Expensive operations, such as bytecode verification, need not be repeated because of the strong consistency guarantees of the OPJ model.

Neverthless, there are trade-offs to be made between the speed of resumption and the complexity of an OPJ implementation. For example, it was observed in the Napier88 system [159] that a large number of objects were always faulted in during that system's bootstrap, and that this took a substantial time. Based on this experience, PJama$_{0.0}$ was designed to include a bootstrap region, containing the equivalent system objects, that was read into memory in one go on resumption. Since this data was mostly C data structures, an ad hoc pointer-swizzling scheme was used to relocate the data in memory. This special case greatly complicated activities such as evolution, and early measurements showed no significant performance improvement [116] and therefore it was abandoned in PJama$_{1.0}$. This experience demonstrates that one cannot always extrapolate performance intuitions across different systems.

Similarly, PJama$_{0.0}$ saved the optimised "quick" bytecodes in the store, again on the assumption that it was worth avoiding redoing this on resumption. The advent of JIT compilers quickly made this issue moot from a perfomance perspective, but we still lived to regret the added complexity when implementing evolution. Therefore, in PJama$_{1.0}$, we chose to save neither optimised bytecodes nor JITed code in the store. In fact we go further and store the classfiles in their standard format and reload them as needed on a resumption.[1]

However, since JITing code is relatively expensive, particularly as the sophistication of the optimisations increase, we expect to revisit this decision. We have made provision for storing such architecture and application-specific data in our representation of classes in the store, but without compromising the architecture-neutral information.

Clearly there is a trade-off between the time it takes to read such pre-computed information from the store and the time it takes to regenerate it from other forms. Since the relative performance of processors and disks continues to widen, it only seems worth storing information that is very expensive to compute. For example, certain global optimisations might fall into this category.

A final factor in the performance of resumption is the execution of resume listeners, typically to re-establish a connection to an external system. The OPJ specification provides for a global set of listeners that are invoked on resumption ($\rightsquigarrow$5.5). They may be invoked in parallel with each other and with the main application, but they will all be invoked, even if the application may not use the service that they support. The latency of resumption is therefore dependent on the number of registered listeners. It is possible to conceive of cases where this latency might be quite large, although we have encountered no examples in practice. However, we have investigated and prototyped "lazy" listeners, that are invoked on first active use.[2] Lazy listeners finesse the latency problem, but complicate the PJama implementation, which is why we have kept them out of the OPJ specification until we are persuaded that they are really necessary.

12.2.2 Normal Execution Performance

Ideally no cost would be paid for persistence once the working set of objects needed by an application have been faulted into main memory. Some systems [191, 207, 166, 71, 133, 101] approach this goal by utilising the memory management hardware to detect both non-resident objects, and mutated objects for checkpoint purposes. It is currently the case that the cost of servicing the faults, when they do occur, can be very high with existing commercial operating systems, such that the overall cost may be higher than software-based approaches. However, the ability to run as fast as possible when objects are resident might be important in some applications. A related performance goal that is sometimes suggested is that code that is working with non-persistent objects should incur no overhead. Unfortunately this is extremely difficult to arrange with orthogonal persistence since the requirement for persistence independence ensures that this is not a static property of the code. Only a very sophisticated global optimizer could hope to realize such a goal. Furthermore, the goals of continuous computation inevitably reduces the number of non-persistent objects to a small subset.

[1]We only transform the class into the internal representation; we do not redo verification or initialization. Transforming the classfiles into a separate memory representation arguably violates the principles of orthogonal persistence. There are two mitigating explanations. First, the memory structures are C structures in PJama$_{1.0}$, so cannot avail themselves of the support for orthogonal persistence. Second, if they were Java object structures, we would need to solve a tricky bootstrap problem.

[2]The main reason for lazy listeners is to manage class resumption ordering in legacy code.

The PJama prototypes use software read barriers to handle object faults and software write barriers are used to detect mutations. PJama$_{0.0}$, which ran only with interpreted code, achieved an average slowdown of 15% [116]. With the advent of JITed code, our intuition was that the barriers would be relatively more expensive. However, we still see only an average 15% slowdown. Optimisation experiments [32, 168, 102] suggest that this figure could be substantially reduced, but we have not yet integrated such optimisations into a deployed platform. We believe that we could reduce the overhead to an average 5%, which would effectively satisfy the goal of no noticeable overhead.

The above discussion assumes that the working set of objects will fit in the main-memory object cache. For applications for which this is not the case, or for applications which have access patterns that cause high turnover in the object cache, performance is obviously lower, sometimes dramatically so, because of the relative performance differences between primary and secondary memory.

Assuming that an application that can operate within the available virtual address space on a 32-bit machine, it is important to compare the object-grained object-cache management of PJama against the page-grained cache management of the operating system. Many standard Java applications that use other persistence mechanisms use "big-inhale" techniques to build up a large transient heap and rely on the operating system paging mechanisms to manage main memory. Such large heaps also place a considerable strain on the garbage collector, which can interact poorly with the paging system. In theory, PJama can do better, because the main-memory garbage collector operates only on the resident subset of the total object space. In particular, if the actual working set of such an application is only a subset of the total object space, PJama can run such an application with a much smaller heap. While we have experienced such positive effects with some experimental applications, we have not yet made a systematic comparison between operating-system paging mechanisms and the PJama object-cache management. We are currently gathering such measurements and will report on them in a subsequent paper. We should acknowledge that PJama actually reduces the amount of address space that is available to an application because of the additional data structures needed to manage persistence. Therefore, with applications that perform very large bulk loads of data from an external source, PJama can exhaust virtual memory before the equivalent non-persistent virtual machine, unless the application checkpoints frequently. This would be fixed if PJama implemented a steal policy for new or modified objects from the object cache.[3] This is a known limitation of the current system.

For those applications whose total object volume would exceed the available virtual address space on a 32-bit machine, PJama provides an advantage over a standard virtual machine by making it possible to run them, provided that the object space can be built up incrementally in a number of runs. Performance comparisons in this case are only possible against other persistence mechanisms with similar capabilities [117]. The main issues are concerned with clustering, pre-fetching and other techniques to minimise disk traffic. With PJama$_{1.0}$ and our current set of applications, we are just beginning to reach the point where such measurements are possible. The emergence of 64-bit machines and associated operating system support will remove the advantage that PJama offers and it will be a long time before persistent stores push the 64-bit limit. However, 64-bit architectures will put more pressure on the object cache management system, partly because object references, which are prevalent in object-oriented applications, occupy twice as much memory, and partly because an application's virtual address space can easily grow far beyond the real memory available.

12.2.3 Checkpoint Performance

Unlike resumptions, we expect checkpoints to occur frequently. Indeed, in a complete OPJ implementation including persistent threads, checkpoints could be managed by a background thread, with a frequency chosen by the application, trading off performance against loss of work in the face of a crash.

[3]Not to be confused with the buffer-steal policy in the Sphere store layer, which is operational.

A checkpoint should, therefore, have low latency. We should note the similarity between this goal and the related one for garbage collection, and observe that their implementation has many aspects in common. There are two main facets to the performance of a checkpoint. First, the processing time required to identify the mutated objects and the new objects that need to be promoted. Second, the time taken to transfer them to the persistent store with a guarantee of durability.

There is a trade-off between the time that it takes to detect mutated objects and the work that is expended during normal processing to record when an object is mutated. In PJama$_{0.0}$, auxiliary data structures were used to record mutated objects, and these provided an exact and efficient way to determine the modified set. In PJama$_{1.0}$, we chose to minimize the effect on executing code, partly for simplicity and partly because of of the impact on JITed code. The consequence is that we have to scan the entire Resident Object Table at a checkpoint to find the mutated objects. This clearly has scaling problems and we are beginning to see some evidence of long checkpoint latencies in very large applications, which may require us to alter our design and shift the trade-off.

The OPJ semantics does not permit an explicitly called checkpoint to return until the data has been made durable in the store. This requires some synchronous writes to the log, but allows other I/O to take place asynchronously with the mutator, provided that the write-ahead log invariant and the separation between the object cache and the store buffers are maintained.

12.2.4 Measurement Experiments and Preliminary Results

Our measurement experiments to date have, regrettably, been limited, because our primary focus has been on completeness and reliability. However, in our defence, we must note that all of the PJama prototypes have had excellent performance compared with the competition, so we have not been under outside pressure to improve performance substantially.

Early performance measurements with PJama$_{0.0}$ were carried out on several applications, including the Java compiler, a web-server written in the Java programming language, and a software configuration management benchmark from the JP project translated from C++ into Java programming language. These all showed nominal execution overheads within the 15% bound and improvements in resumption time due to the continuous computation model of OPJ.

More recently we have reported on the performance of PJama$_{1.0}$ with the OO7 benchmark [40], the Portable Business Objects benchmark, pBOB [74], and the SpecJVM [193] benchmarks. These continue to show good performance. For example, the throughput overhead on pBOB is only 9% for PJama$_{1.0}$. In addition, PJama$_{1.0}$ can support pBOB configurations that exceed the virtual memory limits. We have built stores of 5 GBytes in size that operate effectively in a limited amount of real memory.

We are in the process of preparing a report [117] on a representative sample of persistence mechanisms for the Java platform, compared according to the criteria established in $\rightsquigarrow$3.2.

12.2.5 Measurement Summary

We have only just finished constructing PJama$_{1.0}$ and there has been little time to tune its algorithms and to remove bugs that interrupt measurements. We consider it important to carry out systematic measurements to validate and improve our design. The experiments so far attempted suggest that performance will be at least acceptable. But such a claim is susceptible to criticism, until we have an established set of benchmarks and protocols for using them that correspond with the kind of continuous, sophisticated data and computation intensive applications that we envisage. It is important that any assessment eventually includes: concurrent disk garbage collection, evolution, migration, archiving, durability and endurance, Challenge 14 on Page 77.

Chapter 13

Related Work

When the PJama project began the Java platform (JDK 1.0.2) provided very little support for persistence. It included the traditional mechanisms to encode and decode basic types using input/output streams that could be connected to files in an external file system. In addition, there was ad hoc support for encoding and decoding a property table (`java.util.Properties`) to and from a stream. However, the language designers were aware of persistence, as they provided a declaration modifier, **transient**, that was intended to distinguish fields that were not "persistent". Unfortunately, the language specification initially did not specify the semantics of this modifer, leaving it open to subsequent clarification.

The JDK 1.1 release added Java Object Serialization (JOS) and Java Database Connectivity (JDBC) [97]. JOS is a mechanism that supports the encoding of an object to an output stream and the subsequent decoding from an input stream. JOS is used both for persistence and for communicating copies of objects in the support for distributed programming with Java RMI ($\rightsquigarrow$11.1).

Unlike the property table encoding, which is textual, JOS uses a binary encoding format. JOS is customisable by allowing a class to override the default encoding by providing special methods that can perform arbitrary transformations of the object format during the encoding/decoding process. Unfortunately, the transient modifer was misused by this mechanism. Instead of the interpretation of "not persistent", suggested by the JLS, it was interpreted by JOS as meaning "possibly custom serialization". There are many examples of this idiom in the standard libraries, for example, the fields denoting the content of a `java.util.Hashtable` are marked **transient**. This idiom renders **transient** unusable for orthogonal persistence [176].

JOS is effectively the default persistence mechanism for the Java platform, and is used extensively in the JavaBeans^{TM} framework. However, difficulties with managing the co-evolution of classes and associated serialized data, have led to a gradual disillusionment, which has led to a new proposal for JavaBeans persistence [151]. From an orthogonal persistence perspective, the source of these problems is the lack of completeness in JOS, in that the bytecodes for the classes are not included in the serialized form, thus rendering certain kinds of evolution impossible. JOS also suffers scalability problems because a serialized form must be read in its entirety (the "big-inhale" problem), before any object can be processed by the application. Attempts to avoid this by separating the data into multiple streams, encounter equally difficult problems managing the coherence and relationships between the individual, but not independent, streams.[1]

JDBC provides a standard way to communicate with a relational database using the SQL language [97]. For simple applications, JDBC is quite easy to use and it has been phenomenally successful. Neverthless, as application complexity grows, JDBC becomes harder to manage, particularly for applications that wish to operate at the object level. Two solutions have been developed to attack this problem. The first, SQLJ [46], is a traditional embedding of SQL within a Java application.[2] A pre-processor transforms a SQLJ program

[1]Note that XML-based solutions will suffer similar problems.

[2]Except that the strong typing in the Java programming language has been exploited to help programmers use cursors correctly.

into the equivalent Java application with explicit JDBC calls. A characteristic of the SQLJ approach is that the dominant data model is SQL, with the Java programming language operating as the host language in SQL terminology. This suffers from the impedence mismatch which orthogonal persistence sets out to eliminate.

The second solution is referred to generically as object-relational mapping (OR-mapping).[3] In contrast with SQLJ, OR-mapping attempts to make the Java object model dominant, by hiding the relational data model in a layer of generated transformation code. Typically, OR-mapping can map existing relational data models into an object model or vice versa. Since this mapping can be quite complex, sophisticated tools are provided to assist the developer in managing the mapping. The JavaBlend^{TM} system [202] is an example.

The Object Database Management Group (ODMG) has defined a binding of its object model to the Java programming language [42] ("ODMG Java binding"). In doing so, they have brought their model closer to orthogonal persistence: adopting persistence by reachability, allowing language definitions to determine a schema and accommodating nearly all Java class libraries. Many object database vendors have implemented the ODMG Java binding to the Java programming language. Since the mapping is closely aligned with the Java programming language, such systems provide a fairly straightforward route to object persistence for the Java programming language. The devil, unfortunately, is in the details. The ODMG Java binding allows many implementation-specific behaviours that fall some way short of the principles of orthogonal persistence, particularly with respect to persistence independence which they view as infeasible. As a result, it is generally impossible to reuse a Java class library unchanged in an ODMG compliant system. In addition, it is frequently the case that either pre-processing of source code or post-processing of classfiles is required, which greatly complicates the development process, and does not permit the full generality of the dynamic class loading facilities to be utilized. In this group Gemstone/J stands out as providing the closest approximation to orthogonal persistence, and it is not just coincidence that Gemstone/J also shares the PJama approach of modifying the JVM [35, 87, 88, 34].

At the time of writing ODMG has suspended development of the ODMG Java binding pending completion of the Java Data Objects (JDO) specification that is being developed under the Java Community Process [187]. The majority of the ODMG members are involved in the JDO specification, but the JDO interest group also includes relational database and OR-mapping vendors. JDO is an ambitious attempt to define a standard API that provides access to a wide variety of heterogeneous data sources. The intent is that these sources can be manipulated by a Java application as if they were simply ordinary Java objects. An application may access objects from multiple JDO sources concurrently. In principle, JDO subsumes OR-mapping or, more accurately, an OR-mapping system would appear to a Java application as simply another JDO source.

The relationship between the JDO (and the ODMG Java binding) and OPJ is quite subtle. It is possible to argue that JDO subsumes OPJ and, indeed, this argument was used to reject the OPJ specification request in the Java Community Process. Superficially this argument is appealing but, as with the ODMG Java binding, the differences between OPJ and JDO are in the details, and the crux of the debate is about whether these details are important. We can illustrate this by noting that, in OPJ, the standard memory model defined in the JLS is in effect, with the sole addition that durability of the memory is controlled by the checkpoint operation. JDO, in contrast, defines a complex and separate "memory model" that the programmer must understand. For example, this model includes the concept of a "hollow" object, which is one that exists in the virtual machine but whose content has not yet been loaded from the external data source. Programmers must be prepared to deal with hollow objects in certain circumstances. In contrast, while an OPJ implementation must also create objects in the virtual machine in response to object faults and copy the contents from the persistent store, an object in an incomplete state is never visible to the programmer. We can relate this difference to one of our architectural choices, namely the 1-1 relationship between a PJVM and a persistent

[3]This is not to be confused with object-relational databases.

store. JDO permits a many-many relationship with two consequences. First, that a transaction API must be used to delimit access to JDO-managed data and, second, that JDO must resynchronize with the external store whenever a transaction begins, because another PJVM may have interleaved its access to the same store.

Finally, the Enterprise Java Beans (EJB) framework [200] aims to provide a complete solution to applications that manage distributed, heterogeneous data. Unlike JDO, EJB makes no pretence of persistence independence, programmers must follow strict rules about the structure and form of the classes that make up an EJB. In fact, each bean instance is represented by a total of three classes, the “EJB object”, the “EJB home” and the actual bean instance. All access to the bean instance is routed through the EJB object and completely controlled by the EJB server. The benefit from following this protocol is that the EJB infrastructure promises to automate many aspects of persistence and transaction management. In addition, the EJB server can provide instance pooling and object-cache management with similar goals to the object-cache management provided by PJama. It is fairly easy to see how this framework can support simple business applications, but it is unclear what the performance overheads will be or whether object-cache management can operate efficiently enough at this level. It seems less likely that EJB can extend beyond simple business applications into the realm addressed by orthogonal persistence, namely applications with complex, shared and long-lived object data, given the levels of indirection and substantial amount of boilerplate code generated per bean.

Chapter 14

Conclusions and Future Work

Here we present a summary of the facts elicited by reviewing the PJama project, and then revisiting the major decisions in the light of those findings (⇝15). This leads us to conclude with a list of research issues that we consider worthy of investigation (⇝16).

14.1 Summary of the State of the PJama Platform

The latest release, PJama$_{1.0}$, provides a well developed framework and technology for further orthogonally persistent platform and object store research, and for experiments into the utility, viability and performance of orthogonally persistent platforms (⇝2). Powerful facilities are presented via a deceptively simple API (⇝5). It is supporting substantial application development (⇝12.1): the largest stores we've observed so far have been approximately 5 gigabytes; the largest population of objects we've observed has been 60 million; and the largest single object, about 34 megabytes. We believe that it will scale well beyond this. It meets aspects of every one of our requirements (⇝3.2). This has required balancing investment of effort (⇝A) and trading-off performance (⇝12.2). Each requirement has either been met or we have well advanced research into how it will be met.

A **Orthogonality** This is complete for all application-programmer-defined classes, for all classes defined entirely in the Java programming language and for all classes that use native code but comply with the Java Native Interface (J.N.I). It also includes class `Class`. The current deficiencies are classes `Thread` and `Exception` which are intimately interconnected with the JVM implementation (Challenge 3 on Page 75)[1] and some core classes with external state that have not yet been adapted to the resumption programming model (⇝5.5).

B **Persistence Independence** This is complete apart from two temporary work-arounds for: the misuse of the **transient** modifier (⇝5.4) and `Thread` not being persistent (⇝12.1 ❹).

C **Durability** This is complete apart from incremental archiving (⇝7.3).

D **Scalability** In the range from palm tops to 5 gigabytes, we have encountered no fundamental difficulties, and we expect to extend this to 100 gigobytes or more (⇝7).

E **Schema Evolution** A comprehensive schema evolution system with good safety properties is operational (⇝8).

[1]There are versions of PJama that do not suffer this lack of orthogonality, but they sacrifice performance.

F **Platform Migration** This is currently achieved by logical archiving (⇝9).

G **Endurance** There are two aspects of this: the reliability of the implementation and the ability to carry out all essential administrative tasks concurrently with the workload. Considering the former, we can exhibit runs of a few hours to a few days. Considering the latter, there are still functions to implement.

❶ Disk garbage collection, including cycle collection.

❷ Archiving, and less urgently, logical archiving for migration.

❸ Evolution, including deployed evolution (⇝8.2).

❹ More ambitiously, dynamic self-tuning and reorganisation.

H **Openness** Our model and API support this completely. As explained in (A), the implementation is incomplete; the principal classes not yet handled are: `java.jdbc` and `java.corba`.

I **Transactional** The well established implicit transactions with checkpointing are available and research is well advanced into powerful support for controllable combinations of isolation and sharing in the whole computation (⇝10).

J **Performance** When the total workload is considered, including interacting with long-lived data, our performance is generally significantly better than other systems. Certainly we have encountered no "showstoppers" though we still discover algorithm pathologies as we expose new aspects of scale, such as a large volume of updates or a checkpoint with tens of millions of new objects.

In other words, we have shown that the technology is feasible by producing a working demonstration. The deficiencies are those typical of any new software architecture at this stage of maturity. In some cases, such as concurrent disk garbage collection and concurrent evolution, our technology is already well in advance of many commercial products. We believe our platform has a design which will enable the implementation of endurance to be completed.

We have explored the task of building an orthogonally persistent platform according to a particular set of decisions (⇝15). This has led to a considerable number of insights and results reported throughout this report and in our cited papers. We would encourage others to explore platforms that meet similar requirements by making these choices differently and to join us in developing and using measurements to establish the best practices in engineering these platforms.

The basic tenet that drives this research is the belief that it is essential to invest in programming platform infrastructure in order to increase the productivity of enterprise application developers, and the performance and reliability of the applications they produce. Longitudinal studies are necessary to evaluate whether this hypothesis holds out in practice. PJama is available to anyone who wishes to conduct such studies or who wants to experience its convenience for themselves.

Chapter 15

Review of Decisions

A number of the major decisions taken in the PJama project are revisited and assessed. Each is given an identifier for convenient cross-reference.

To use the Java programming language

Decision 1

The alternative was to continue with research languages (⇝2.3) or to have used some other commercial language. The former would certainly have avoided much complexity, would have avoided tracking distracting changes and would have afforded us with a better type system (though that of the Java programming language is expected to improve [31]). It might have allowed us to repair irksome language design flaws. However, none of them would have afforded access to rich libraries and potential large scale trials. The potential value of choosing the Java programming language rather than a research language has yet to be realised, by conducting well-designed orthogonal persistence hypothesis testing. We are not aware of any other, commercially active language that has comparable type safety but less complexity.

To develop based on the JDK

Decision 2

There was no alternative at the start of the project, so this must be recast as the decision to continue with the JDK (⇝6.4). This has cost large amounts of labour (⇝A) handling its complexity and tracking its changes (⇝9.2). So far, the effort of making a transition to a different platform and the hope that by working with the definitive platform we might influence it, have militated against abandoning the JDK. There is also a suspicion that current alternatives would prove equally complex when investigated in detail. This provokes the question: is it possible to arrange that this kind of work is less costly in the future? (see Challenge 2 on Page 74)

To give class `Class` the standard rights to persistence

Decision 3

To restrict class `Class`'s persistence rights (⇝5.2) would not only create an anomaly in the reachability and orthogonality rules, it would also remove the promise that a store is self-describing and consistent. With classes in the store, we can continue the Java programming language commitment that the only operations applied to their instances are the operations permitted by their class and the prevailing security manager. This makes it necessary for PJama to deliver adequate evolution and migration facilities ($\rightsquigarrow$8 and $\rightsquigarrow$9), but it also makes it possible.

TO GIVE CLASS `Thread` THE STANDARD RIGHTS TO PERSISTENCE

Decision 4

We have aspired to give `Thread` the standard rights to persistence. This is possible at present only by restricting our capacity to migrate and evolve, or in inherently inefficient implementations ($\rightsquigarrow$4.2.1, $\rightsquigarrow$5.2, $\rightsquigarrow$8 and $\rightsquigarrow$9), Challenge 3 on Page 75.

TO ADOPT A ONE-STORE-TO-ONE-PJVM ARCHITECTURE

Decision 5

We have focused on an architecture where one virtual machine interacts with one store, which is different from prevailing products ($\rightsquigarrow$6.1). We restrict a PJVM to one store in order to prevent a requirement for multi-store commit protocols and an unmanageable tangle of inter-store references.[1] We choose one PJVM at a time, operating against a store because we believe that this best fits the emerging server architecture of multi-processor, large address space and large memory application servers. We believe it is also the right choice for supporting sophisticated interworking via multi-threading.

TO MANAGE OBJECTS

Decision 6

The platform needs to manage the space on disk, the movement of data to and from disk, and the identification of data on disk. We chose to do this in terms of objects ($\rightsquigarrow$6.2 and $\rightsquigarrow$7.2) rather than pages. The mapping between the current hardware, which is tuned to pages, to the object model has to be made at some stage. We suspect that logical simplicity, flexibility and incremental algorithms flow from this and that they will more than compensate for any performance penalty.

TO CACHE RESIDENT PERSISTENT OBJECTS ON THE `gcheap`

Decision 7

This avoids copying and duplication costs during promotion but adds complexity to the main-memory garbage collector ($\rightsquigarrow$6.6). It has the potential advantages of synergy between the main-memory garbage collector and object eviction, and of dynamic balance between transient and persistent object allocation. But it is very complex to implement (see Challenge 15 on Page 78).

[1]It is not just that this adds to platform implementation costs; it also leads to complex failure modes and operational problems and thereby impairs the simplicity which we aim to deliver to application developers.

Swizzling lazily with exceptions

Decision 8

This reduces the demand for resident object table space, and enables incremental loading and eviction, but it has execution penalties from more extensive read barrier requirements (⇝6.5).

Extensive use of indirection in the persistent object store

Decision 9

The classes are reached via a descriptor (⇝7.1), partitions are reached via a logical-to-physical mapping and objects are reached via an indirectory (⇝7.2). This has a cost in additional disk latency only partially overcome by caching, but permits localised reorganisation, concurrent space management, incremental evolution algorithms and efficient logging. Alternatives to some of these indirections must be explored to accurately assess their costs and benefits.

Division of the persistent object store into partitions

Decision 10

The advantage of a partitioned store is that incremental management algorithms are more easily constructed and may be subject to different management regimes (⇝7.2). It has a potentially high cost in inter-partition structures and their management. However, it is possible to trade between these to some extent by adjusting the size of partitions in response to the interconnectivity of stored data and the workloads experienced.

Attempting completely general evolution

Decision 11

The evolution of classes and reformatting of instances is as complete and general as we can make it (⇝8). This is considered necessary, so that data can always be transformed to meet new needs without loss. On the other hand, it has a high cost of requiring the support of developer-defined code during an evolution.

Safety checking evolution

Decision 12

We choose to attempt to validate a proposed evolution before performing it (⇝8.3) instead of trusting programmers. We believe that the complexity and processing this adds is more than recompensed by the reduced risk of rendering a store unusable and by the value to developers of its error reports.

Discrimination between distribution and persistence

Decision 13

We choose to retain the explicit approach of Java RMI for distribution rather than faulting in remote objects, in the same way as we fault in persistent objects (⇝11). This may be less simple, but it has considerable operational advantages, and it is commensurate with the higher rates of failure and partial failure experienced with distribution. Essentially, a federated model was favoured over a distributed shared memory model.

Chapter 16

Challenges and Opportunities

We draw together the major research challenges and opportunities identified during this review, in a similar style to the decisions ($\rightsquigarrow$15).

CONDUCT EFFECTIVE ORTHOGONAL PERSISTENCE TRIALS

Challenge 1

This is perhaps the most fundamental challenge. All others derive from our attempts to try to meet it. Enterprise applications involve huge investments of skilled software engineering labour. It is postulated that this could be more efficiently deployed by using orthogonal persistence in conjunction with whatever other enterprise platform improvements are developed. This expectation has not been independently tested on realistic application development and deployment projects that have sufficient scale, complexity and duration. The design, conduct and analysis of such a trial is itself a research issue. This needs to be completed and its results fed back into enterprise platform development before the currently emerging technology reaches its limits. At present benchmark results dominate customer and researcher choices. Not because speed is the ultimate issue in many applications, but because these metrics can be transparently compared. We believe that this is distorting the enterprise platform industry: inappropriately depleting the effort invested in ease of development, deployment and evolution. These require the establishment of correspondingly transparent and accepted metrics (see Challenge 14 on Page 77). We have learnt that software structure, and more particularly programmer habits, are deeply imbued with the patterns demanded by database and file-based models of persistence. Consequently, the change to orthogonal persistence is deeply disruptive at the enterprise system scale. Therefore, before practitioners will contemplate a switch to the new technology there will have to be compelling evidence.

SIMPLIFY THE PERSISTENCE IMPLEMENTATION TASKS

Challenge 2

General research into the implementation of platforms, such as the Java platform, may yield improvements in the structure of the platform and its efficiency. However, observation of new platforms, e.g., Java HotSpot technology [10], Jalapeño [6], Open-JIT [170] or IBM's JIT [199] suggests that much complexity remains, and that it may actually increase as implementers strive for performance.

It may be possible to influence such efforts, so that either persistence is considered from the start of design, or more generally to specify a set of hooks (interfaces) that a virtual machine should include, so that persistence can easily be added. Just defining and validating such PJVM hooks, would assist future language designers and implementors to include persistence at reasonable cost. It would probably take the form of separate barrier instructions, that could be implemented in various ways, including as null operations. Optimisers would remove redundant cases, and code generators or interpreters would implement their specified semantics. Identifying these barriers and incorporating them into architecture-independent PJVM specifications would be a major step forward.

In the case of the Java platform, much of the effort is in handling the core library (⇝9.2). It would be a worthwhile exercise to show how the work on core libraries could be minimised by recoding entirely in the Java programming language and by using a resumable-programming style (⇝5.6) where ever it is appropriate. Note that this demonstration could be done for a single sublibrary, such as a GUI library.

DEVELOP A MODEL FOR PERSISTENT THREADS

Challenge 3

This challenge (introduced by Decision 4 on Page 72) requires a thread model that is still capable of exploiting hardware, such as multiple processors, and yet can be made persistent. Once persistent, it may be resumed at a later time, in a different context, for example, on a different hardware or virtual machine platform. Not only must it withstand migration (⇝9), but it must also provide sufficient reflection that evolution can verify consistency and make changes if necessary.[1] Similarly, its structure must be described for disk garbage collectors, as objects reachable from persistent active threads must persist. Clean semantics and the potential for efficient implementation on the full range of platforms are both required. We suspect that this model will also be of interest to interactive debugging interfaces and distribution research.

DEFINE THE SEMANTICS OF ENDURING BUT OPEN COMPUTATIONS

Challenge 4

Our interest in this challenge is explained in ⇝5.1 but we suspect that it may be of much wider interest. An enduring computation is one that continues indefinitely. It is expected to yield correct outputs in accordance with its stream of inputs so far (which must determine the current state of any persistent store and the current versions of all code to be executed) and take into account the autonomous changes in its environment (whose state is only known when it is sensed) e.g., the availability of a communication channel or the current state of a hot swappable disk configuration. We suspect that this semantics is not only necessary for all platform code (e.g., operating systems, database servers, enterprise servers, etc.) but also for much of the computation that supports our current life-style: software to control devices, telephone networks, etc. We note that there is some work emerging in this area [216].

DEVELOPMENT EVOLUTION

Challenge 5

We have established comprehensive mechanisms for development evolution (⇝8.1). Application developers require these facilities to be well integrated with their development tools, such as version and configuration managers, and build managers. It is possible that development tools, such as profilers, assertion checkers,

[1]Self had a mechanism for re-arranging stack frames if a method was changed via the debugging interface [100].

debuggers, selective tracers and animators will use the evolution technology to insert and remove diagnostic code. Essentially, once the provision of persistence is orthogonal and implicit, the developer uses the persistence platform at every step. Both the new potential exposed by integrated persistence and the new requirements that it imposes await exploration.

DEPLOYED EVOLUTION

Challenge 6

Once orthogonally persistent platforms are in use, the deployed systems will be housed in stores containing both instances of objects and the code that defines their behaviour ($\rightsquigarrow$8.2). The customers of these systems will certainly expand the population of instances and may extend the set of classes. They will expect to receive new versions of the provided software, while retaining their investment. This requires the development of new technology (a) to extract and analyse the cumulative developments that must be sent to each customer and (b) a mechanism for safely and non-disruptively installing the upgrades.

CONCURRENT AND CONTROLLED EVOLUTION

Challenge 7

At present our evolution technology ($\rightsquigarrow$8) uses eager off-line algorithms. It therefore takes all of the contents of a store from the unevolved to evolved state in one step. As applications grow in scale, as we support endurance, and as multiple applications start to co-exist in one store, two advances are needed (a) the evolution must take place without disrupting service (requiring lazy, incremental and concurrent algorithms that are still atomic and scalable) and (b) the scope of evolution within the store must be defined (in order that multiple versions of classes can co-exist, as different customers using the same store will wish to choose independently when to adopt new versions).

CONCURRENT DISK GARBAGE COLLECTION

Challenge 8

As increasing scale and increased endurance both grow in importance, it becomes vital to support complete, safe, and scalable disk garbage collection concurrently with mutator loads. Achieving all of these efficiently is still a research challenge.

A VALIDATED ENGINEERING MODEL OF PERSISTENT PLATFORMS

Challenge 9

The design and operation of persistent platforms requires adequate models of performance and implementation cost ($\rightsquigarrow$6). These models should provide an engineering basis for architectural design decisions and a framework for a self-tuning system. They should be validated against real loads at the scale to which they apply. Fortunately, this challenge can be addressed incrementally, by better understanding and modelling of some subsystems, provided the models can be combined. This in turn requires the community to establish a common framework for this modelling. The research has to include the characterisation of workloads. An example is given by storage system research [186, 208, 30].

Well-tuned and Self-tuning Persistent Object Stores

Challenge 10

The flexibility to have multiple representations, multiple store administration schemes and many controls over sophisticated algorithms have been introduced ($\rightsquigarrow$7). This exposes a plethora of options that are intractable without a systematic approach. In the short term, analysis of store behaviour is required to permit initial choices and settings to be made. Research is necessary to achieve storage schemes that are sufficiently easy to install and manage, and which adapt to the variation of workloads.

An Architecture for Very Large Persistent Object Stores

Challenge 11

Much larger object stores, tens of TBytes and beyond, will require new architectures to permit sufficient parallelisation of object traffic and store management applications ($\rightsquigarrow$7). These architectures should be well informed by large-scale existing systems and hardware trends. The cost of a new store architecture is high. The challenge is to move to such a new framework with reasonable certainty that it will provide the expected performance, flexibility, longevity, scale and opportunities for new mechanisms.

Effective Isolation and Interworking within a Virtual Machine

Challenge 12

Applications are composed of many sub-applications and require to interact safely but rapidly with concurrently executing applications ($\rightsquigarrow$10.3). The low-level technology available looks very promising ($\rightsquigarrow$10.1 and $\rightsquigarrow$10.2). Support of a complex transaction model designed to support collaborative working has already been demonstrated ($\rightsquigarrow$10.3.2). The challenge remains to complete this provision of safety with interworking and to integrate it with high-level notations and structures that aid maintenance and evolution.

Developing viable and usable combinations of Persistence and Distribution

Challenge 13

There is considerable technical and intellectual difficulty in finding a combination of persistence with distribution ($\rightsquigarrow$11) which supports developers well, is scalable and is compatible with class evolution.

Establishing Appropriate Benchmarks and Metrics

Challenge 14

A new kind of application system is envisaged — large scale, evolving, long-running, complex data, complex software, etc. New patterns of computation, evolution, migration and endurance need to be captured. The measures must look at long-term and appropriate balances between these, characteristic of real applications. Finally, it would be ideal if complexity metrics could be introduced so that performance could not be bought by hand-crafted contortions in the application code. This interacts with Challenge 9, but is not the same. Here we seek the equivalent of benchmarks, such as Specmarks and Transaction Processing Performance Council benchmarks C (TPC-C) and D (TPC-D) [103], but we require the benchmark to be concerned with *software engineering costs* in a proper balance with throughput, reliability, recovery, migration and evolvability.

GOOD ENGINEERING FOR INTEGRATING OBJECT CACHING AND GARBAGE COLLECTION

Challenge 15

It is difficult to provide high-performance garbage collectors, and the rest of the main-memory space management system cpable of carrying the loads imposed by all application programs. Similarly, the development of good object-caching algorithms, including effective eviction, requires careful engineering. The challenge is to understand how to build these in combination. It is possible that they might operate synergistically. For example, the garbage collector might collect information for eviction algorithms and help with patching references to evicted objects, and the object cache might help the garbage collector by storing inactive objects on disk. But these both remain complex topics, so the first step is to develop appropriate interfaces between them.

Acknowledgements

This work was undertaken jointly by the Forest team at Sun Microsystems Laboratories and the University of Glasgow. The Glasgow team received support from the University, from the British Engineering and Physical Science Research Council (GR/K87791) and from Sun Microsystems, Inc. The principal team members who developed PJama were Grzegorz Czajkowski, Laurent Daynès, Misha Dmitriev, Neal Gafter, Craig Hamilton, Brian Lewis, Bernd Mathiske, Tony Printezis, Susan Spence and Michael Van De Vanter. Many students at Glasgow and several summer interns at Sun, contributed in various ways. Laurent Daynès, Mario Wolzcko and Jeanie Treichel read drafts of this paper and provided helpful comments.

Appendix A

Investment of Effort

We can estimate our effort in terms of engineer-months, as shown in Table A.1 where additional student months are shown in brackets.[1] The following table sets out to do this for each calendar year and for the ten requirements ($\leadsto$3) and the application and development work ($\leadsto$4). In this analysis, all workers in each category are considered equal. The allocation of work-to-requirements categories is inevitably highly subjective. It is explained further in the notes below the table. Note that we started in September 1995, and that the time in that year includes design work and initial study of the JDK. The column for the year 2000 only refers to the period from January to April.

		Engineer (student) Months						
Requirements		1995	1996	1997	1998	1999	2000	Total
A	Orthogonality	8	4	2	2	5	2	23
B	Independence	8	8	3	3	3		25
C	Durability		1	5	14	15	2	37
D	Scalability		6	2 (5)	13	12	2	35 (5)
E	Evolution			3	14	20	4	41
F	Migration		9	12	26	21		65
G	Endurance		3	12				15
H	Openness			8	10	3		21
I	Transactions	1	1	2	12	18	6	40
J	Performance		10	6	6	20	1	43
K	Evaluation		12	15 (7)	17 (21)	32 (30)	6	82 (58)
Total		17	54	70 (12)	117 (21)	149 (30)	23	427 (63)

Table A.1: Table showing Effort Allocation from 1 October 1995 to 1 April 2000

Some aspects of this table are discussed below.

A *Orthogonality* was well understood prior to the work. The effort was initially a matter of design, and implementation of a platform to handle correctly everything written in the Java programming language. The incremental work continues because of the need to deal with classes that have been

[1]This includes the team members at Glasgow in Malcolm Atkinson's group and at Sun in Mick Jordan's group (including contractors), and an estimated contribution from Quintin Cutt's group at Glasgow, and Tony Hosking's group at Purdue. Staff members and PhD students (whether in their university or interning at Sun) were counted as engineers. Only good quality MSc and experienced undergraduate work was counted as student work.

implemented using C. Those that interact with the platform implementation, `java.lang.Class`, `java.lang.Thread`, etc. require significant effort. The introduction of JNI into all of the C code in core classes led to improved efficiency in this aspect of our work, as most of the code could be handled by introducing read and write barriers into the JNI implementation.

B *Persistence Independence*: The main effort here was to change correctly the implementation of the JVM so that it provided persistence but still complied with the JLS. Developments to the JLS and to the JDK require us to revisit this work.

C *Durability*: Initially we only had to map to Recoverable Virtual Memory and to repair its bugs. Once we pursued a scalable solution based on logging, there was effort involved in designing and implementing a version of the ARIES algorithms [156] ($\leadsto$7.3).

D *Scalability* includes the work on scaling up our store capability ($\leadsto$7), in relieving main-memory constraints (PJama$_{1.0}$), and in exploring small systems (PJama$_{pt}$).

E *Evolution* implementation began afresh in 1998 ($\leadsto$8) after an initial experiment on class consistency verification in 1997.

F *Migration*: Grouped under this heading is all the work that we have had to do to allow developers using PJama to track the releases of successive versions of the Java platform as exemplified by the JDK. Section 9 discusses this work and contains Table 9.1 which illustrates the scale of the problem. This is an area that we seriously underestimated. We had not expected such a large volume of changes and extensions per year. It also contains our early work porting the platform to WINDOWS and our change over to the SRVM architecture in order to gain the benefits of JIT optimization and exact memory management.

G *Endurance*: There was an initial investment in disk garbage collection and cache eviction. This has been revitalised using the PJama$_{1.0}$ technology, and work is in progress on disk garbage collection running concurrently with an active load.

H *Openness*: Pioneering work on resumable programming and its application to sockets and Java RMI terminated once that was operational, though the programming model has been refined since. Refinements and alternatives have been explored. The challenge is to meet all of the contingencies thrown up by legacy Java applications while retaining an easily understood programming model.

I *Transactions*: The simple transactional model, pioneered in PS-algol [13] and Napier88 [159], was straightforwardly incorporated into the design and implemented with Recoverable Virtual Memory. The flexible transaction model ($\leadsto$10) continues to command substantial research investment.

J *Performance*: Initially the work on performance concerned buffer and cache management. A line of research into optimisations to avoid barrier tests [32] has continued through recent years. The last 18 months has seen a significant effort to combine persistence and JIT technology in PJama$_{1.0}$.

K *Evaluation*: The work referenced here, includes most of the application construction at both of the main sites, attempts at comparison with rival technologies, and assessment of the performance and usability ($\leadsto$12). The labour involved in the simple use of the platform by colleagues and students, and evaluation work carried out by any of the 160 licencees, is omitted.

Analysis of where effort has gone indicates that a great deal has been concerned with managing the complexities of the Java programming language and the JDK implementation. For basic research, e.g., how to incorporate or implement transactions, these complexities are a serious distraction. Indeed, battling with these complexities and coping with the rapid rate of change in the Java platform are, in themselves, of very little research value. The value of this effort will only be realised when we are able to observe and evaluate real enterprise applications running on an orthogonally persistent platform because we have persevered with this battle to achieve a viable platform. This is discussed further in $\leadsto$15 and $\leadsto$16.

Bibliography

[1] R. Agrawal and N.H. Gehani. ODE (object database and environment): the language and the data model. *ACM SIGMOD Record*, 18(2):36–45, June 1989.

[2] R. Agrawal and N.H. Gehani. Rationale for the design of persistence and query processing facilities in the database programming language O++. In Hull et al. [104], pages 25–40. Proceedings of the Second International Workshop on Database Programming Languages (Salishan Lodge, Gleneden Beach, Oregon, June 1989).

[3] C. Ahlberg and B. Shneiderman. Visual information seeking: Tight coupling of dynamic query filters with starfield displays. In *Human Factors in Computing Systems. Conference Proceedings CHI'94*, pages 313–317. ACM, 1994.

[4] A. Albano, G. Ghelli, and R. Orsini. An Introduction to Fibonacci: a Programming Language for Object Databases. In Atkinson and Welland [25], chapter 1.1.2, pages 60–97.

[5] A. Albano and R. Morrison, editors. *Persistent Object Systems: Implementation and Use (Proceedings of the Fifth International Workshop on Persistent Object Systems)*, Workshops in Computing, San Miniato, Italy, September 1992. Springer-Verlag.

[6] B. Alpern, C.R. Attanasio, J.J. Barton, A. Cocchi, S.F. Hummel, D. Lieber, T. Ngo, M. Mergen, J.C. Shepherd, and S. Smith. Implementing Jalapeño in Java. In Meissner [150], pages 314–324.

[7] J. Ambroziak and W.A. Woods. Natural Language Technology in Precision Content Retrieval. Technical Report SMLI TR 98-69, Sun Microsystems Laboratories, M/S MTV29-01, 901 San Antonio Road, Palo Alto, CA 94303-4900, USA, 1998.

[8] O.J. Anfindsen. Conditional Conflict Serializability — an application-oriented correctness criterion. In *International Workshop on Issues and Applications of Database Technology — IADT'98, Berlin, Germany*, pages 47–54, 1998.

[9] O.J. Anfindsen. Conditional Conflict Serializability — an application-oriented correctness criterion. *Database Management*, 9(4):22–30, 1998.

[10] E. Armstrong. Cover story: HotSpot: A new breed of virtual machine. *JavaWorld: IDG's magazine for the Java community*, 3(3), March 1998.

[11] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: A relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976. Also published as: IBM, San Jose, Research Report. No. RJ-1738, Feb. 1976. Reprinted in [197].

[12] M.P. Atkinson. Persistent Foundations for Scalable Multi-paradigmal Systems. In M.T. Özsu, U. Dayal and P. Valduriez (eds.), *Distributed Object Management*. Morgan Kaufmann, 1992.

[13] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[14] M.P. Atkinson, V. Benzaken, and D. Maier, editors. *Persistent Object Systems (Proceedings of the Sixth International Workshop on Persistent Object Systems)*, Workshops in Computing, Tarascon, Provence, France, September 1994. Springer-Verlag.

[15] M.P. Atkinson, K.J. Chisholm, W.P. Cockshott, and R.M. Marshall. Algorithms for a persistent heap. *Software — Practice and Experience*, 13(3):259–272, March 1983.

[16] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java^{TM}. *ACM SIGMOD Record*, 25(4), December 1996.

[17] M.P. Atkinson, M. Dmitriev, C. Hamilton, and T. Printezis. Scalable and Recoverable Implementation of Object Evolution for the PJama Platform. In Dearle et al. [73].

[18] M.P. Atkinson and M.J. Jordan. Issues raised by three years of developing PJama. In C. Beeri and O.P. Buneman, editors, *Database Theory — ICDT'99*, number 1540 in Lecture Notes in Computer Science, pages 1–30. Springer-Verlag, 1999.

[19] M.P. Atkinson, M.J. Jordan, L. Daynès, B. Mathiske, and T. Printezis. A Framework for defining Hooks in the Java Virtual Machine, September 1998.

[20] 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 Connor and Nettles [51], pages 33–47.

[21] M.P. Atkinson, C. Lécluse, P.C. Philbrow, and P. Richard. Design issues in a map language. In Kanellakis and Schmidt [124].

[22] M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3):309–401, 1995.

[23] M.P. Atkinson, M.E. Orlowska, P. Valduriez, S. Zdonik, and M. Brodie, editors. *Proceedings of the Twenty Fifth International Conference on Very Large Data Bases*. Morgan Kaufmann, Edinburgh, Scotland, UK, September 1999.

[24] M.P. Atkinson, P. Richard, and P.W. Trinder. Bulk types for large scale programming. In Schmidt and Stogny [189], pages 228–250.

[25] M.P. Atkinson and R. Welland, editors. *Fully Integrated Data Environments*. Springer-Verlag, 1999.

[26] R. Barga and D.B. Lomet. Phoenix: Making applications robust. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28,2 of *SIGMOD Record*, pages 562–564, New York, June 1999. ACM Press.

[27] S. Berman, R. Southern, A. Vasey, and D. Ziskind. Spatio-Temporal Access in PJava. In Morrison et al. [161], pages 250–258.

[28] E. Bertino, S. Jajodia, and L. Kerschberg, editors. *International Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa, India, September 1996. In conjunction with VLDB '96.

[29] Borland Inprise Inc. JBuilder. http://www.inprise.com/jbuilder/, 2000.

[30] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proceedings of the First International Workshop on Software and Performance*, pages 199–207, 1998.

[31] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 183–200, Vancouver, BC, October 1998.

[32] K. Brahnmath, N. Nystrom, A. Hosking, and Q.I. Cutts. Swizzle-barrier optimization for Orthogonal Persistence for Java. In Morrison et al. [161], pages 268–278.

[33] S. Brandt. *Towards Orthogonal Persistence as a Basis Technology*. PhD thesis, Aarhus University, Department of Computing Science, 1997.

[34] B. Bretl, A. Otis, M. S. Soucie, and R. Venkatesh. Java Virtual Machine Persistent Storage Manager Specification, August 1998. Draft dated 08/26/98 tabled at meeting 09/03/98.

[35] B. Bretl, A. Otis, M.S. Soucie, B. Schuchardt, and R. Venkatesh. Persistent Java Objects in 3-teir architectures. In Morrison et al. [161], pages 236–249.

[36] A.L. Brown, G. Mainetto, F. Matthes, R. Mueller, and D.J. McNally. An Open System Architecture for a Persistent Object Store. In R. Morrison and M.P. Atkinson, editors, *Proceedings of the Twenty Fifth Hawaii International Conference on System Sciences, Volume II, Software Technology, Persistent Object Systems*, pages 766–776, 1992.

[37] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.

[38] G. Canals, C. Bouthier, C. Godart, and P. Molli. TuaMotu: a Distributed Framework for Supporting Enterprise Projects. In *Proceedings of Colloque International sur les NOuvelles TEchnologies de la REpartition (NOTERE'98)*, Montréal, Québec, Canada, October 1998. Editions CRIM. http://www.iro.umontreal.ca/NOTERE/.

[39] G. Canals, P. Molli, and C. Godart. TuaMotu: Supporting Telecooperative Engineering Applications Using Replicated Versions. http://www.idi.ntnu.no/igroup/, November 1998. Internet-based GROupware for User Participation in product development (IGROUP'98), Seattle, Washington, USA.

[40] M. Carey, D. DeWitt, and J. Naughton. The OO7 Benchmark. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Washington D.C, May 1993.

[41] F. Cattaneo, A. Coen-Porsini, L. Lavazza, and R. Zicari. Overview and Progress Report of the ESSE Project: Supporting Object-Oriented Database Schema Analysis and Evolution. In B. Magnusson, B. Meyer, and J-F. Perrot, editors, *Proceedings Tenth Intl. Conference on Technology of Object-Oriented Languages and Systems (TOOLS 10)*, pages 63–74. Prentice Hall, 1993.

[42] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-97 Third Edition*. Morgan Kaufmann, 1997.

[43] M.F. Challis. The JACKDAW database package. In *Proceedings of the SEAS Spring Technical Meeting (St Andrews, Scotland)*, 1974.

[44] M.F. Challis. Database consistency and integrity in a multi-user environment. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 245–270. Academic Press, 1978.

[45] D.D. Chamberlin, M.M. Astrahan, M.W. Blasgen, J.N. Gray, W.F. King, B G. Lindsay, R. Lorie, J.W. Mehl, T.G. Price, F. Putzolo, P.G. Selinger, M. Schkolnick, D.R. Slutz, I.L. Traiger, B.W. Wade, and R.A. Yost. A history and evaluation of system R. *Communications of the ACM*, 24(10):632, October 1981. Reprinted in [197].

[46] G. Clossman, P. Shaw, M. Hapner, J. Klein, R. Pledereder, and B. Becker. Java and Relational Databases: SQLJ. *ACM SIGMOD Record*, 27(2):500, June 1998.

[47] W.P. Cockshott. *Orthogonal Persistence*. PhD thesis, Department of Computer Science, University of Edinburgh, February 1983.

[48] W.P. Cockshott. A Persistent Object Manager for Segmented Virtual Memory. Technical Report ARCH-8-91, University of Strathclyde, November 1991.

[49] W.P. Cockshott, D. McGregor, and J.Wilson. High Performance Operations Using a Compressed Database Architecture. *The Computer Journal*, 14(5):283–296, 1998.

[50] P. Collet and G. Vignola. Towards a Consistent Viewpoint on Consistency for Persistent Applications. In *ECOOP'2000 Symposium on Objects and Databases*, volume 1813 of *Lecture Notes in Computer Science*. Springer Verlag, June 2000. To appear.

[51] R. Connor and S. Nettles, editors. *Persistent Object Systems: Principles and Practice — Proceedings of the Seventh International Workshop on Persistent Object Systems*. Morgan Kaufmann, 1996.

[52] R.L. Cooper, M.P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing Database Systems in a Persistent Environment. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 117–125, 1987.

[53] R. Crawford. Mineyes for TLC, Information Visualisation tools for the Teachers' and Learners' Collaborascope and other generic applications. BSc Dissertation: University of Glasgow, Department of Computing Science, April 2000.

[54] G. Czajkowski. Application Isolation in the Java^{TM} Virtual Machine. In *Proceedings of OOPSLA 2000*, 2000.

[55] S. Daniels, A. Hume, and J. Hunt. Gecko: tracking a very large billing system, 1999.

[56] P. Dasgupta, R.J. LeBlanc, M. Ahamad, and U. Ramachandran. The clouds distributed operating system. *IEEE Computer*, 24(11), November 1992.

[57] C.J. Date and H. Darwen. *A Foundation for Object Relational Database Systems: The third manifesto*. Addison-Wesley, 1998.

[58] L. Daynès. *Conception et réalisation de mécanismes flexibles de verrouillage adaptés aux SGBDO client-serveur*. PhD thesis, Université Pierre et Marie Curie (Paris VI – Jussieu), September 1995.

[59] L. Daynès. Implementation of Automated Fine-Granularity Locking in a Persistent Programming Language. *Software — Practice and Experience*, 30:1–37, 2000.

[60] L. Daynès and O.J. Anfindsen. Implementation of Parameterized Lock Modes using Ignore-Conflict Relationships, 1999. Confidential document SML 99-0602.

[61] L. Daynès and O.J. Anfindsen. Implementation of Apotram Nested Databases using Flexible Locking Mechanisms, 2000. Confidential document in preparation.

[62] L. Daynès and M.P. Atkinson. Main-Memory Management to support Orthogonal Persistence for Java. In Jordan and Atkinson [119], pages 37–60.

[63] L. Daynès, M.P. Atkinson, and P. Valduriez. Efficient Support for Customizing Concurrency Control in Persistent Java. In Bertino et al. [28], pages 216–233. In conjunction with VLDB '96.

[64] L. Daynès, M.P. Atkinson, and P. Valduriez. Customizable Concurrency Control for Persistent Java. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*, Data Management Systems. Kluwer Academic Publishers, 1997.

[65] L. Daynès and G. Czajkowski. High-Performance, Space-Efficient Automated Object Locking, 2000. Submitted to ICDE'01.

[66] L. Daynès and O. Gruber. Nested Actions in Eos. In Albano and Morrison [5], pages 115–138.

[67] L. Daynès and O. Gruber. Efficient Customizable Concurrency Control using Graph of Locking Capabilities. In Atkinson et al. [14], pages 147–161.

[68] L. Daynès, O. Gruber, and P. Valduriez. On the Cost of Lock Inheritance in Lock Managers supporting Nested Transactions. In *Actes des 10èmes Journées Bases de Données Avancées*, Clermont-Ferrand, France, August 1994.

[69] L. Daynès, O. Gruber, and P. Valduriez. Locking in OODBMS clients supporting Nested Transactions. In *Proc. of the 11th Int. Conf. on Data Engineering*, pages 316–323, Taipei, Taiwan, March 1995.

[70] L. Daynès, B. Mathiske, T. Printezis, M.P. Atkinson, and M.J. Jordan. Java Virtual Machine: Hooks for Persistence — views from the PJama team, 1998.

[71] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, 7(3):289–312, 1994.

[72] A. Dearle, D. Hulse, and A. Farkas. Persistent Operating System Support for Java. In Jordan and Atkinson [118].

[73] A. Dearle, G. Kirby, and D. Sjøberg, editors. *Proceedings of the Nineth International Workshop on Persistent Object Systems*. LNCS. Springer-Verlag, September 2000.

[74] R. Dimpsey, R. Arora, and K. Kuiper. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, 39(1), 2000.

[75] K.R. Dittrich and U. Dayal, editors. *Proc. of the ACM/IEEE Int. W'shop on Object-Oriented Database Systems (23rd–26th September 1986, Pacific Grove, CA)*. IEEE, 1986.

[76] M. Dmitriev. The First Experience of Class Evolution Support in PJama. In Morrison et al. [161], pages 279–296.

[77] M. Dmitriev. Class and Data Evolution Support in the PJama Persistent Platform. Technical report, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2000. in preparation.

[78] M. Dmitriev and M.P. Atkinson. Evolutionary Data conversion in the PJama Persistent Language. In *Proceedings of the First ECOOP Workshop on Object-Oriented Databases*, 1999.

[79] I. Dunham et al. The DNA sequence of human chromosome 22. *Nature*, 402:489–496, 1999.

[80] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, California, 1990.

[81] H. Evans. Why Object Serialization is Inappropriate for Providing Persistence in Java. Technical report, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2000. *In Preparation*.

[82] H. Evans and P. Dickman. Zones, Contracts and Absorbing Change: An Approach to Software Evolution. In Meissner [150], pages 415–434.

[83] H. Evans and S. Spence. Porting a Distributed System to PJama: Orthogonal Persistence for Java. In Morrison et al. [161], pages 297–306.

[84] W.S. Frantz and C.R. Landau. Object-oriented transaction processing in the KeyKOS microkernel. In USENIX Association, editor, *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures: September 20–21, 1993, San Diego, California, USA*, pages 13–26, Berkeley, CA, USA, September 1993. USENIX.

[85] A. Garratt, M. Jackson, P. Burden, and J. Wallis. A Comparison of Two Persistent Storage Tools for Implementing a Search Engine. In Dearle et al. [73].

[86] A. Garthwaite and S. Nettles. TJava: a Transactional Java. In *IEEE International Conference on Computer Languages*. IEEE Press, 1998.

[87] GemStone Systems Inc. *GemStone/J Programming Guide*, March 1998. Version 1.1.

[88] GemStone Systems Inc. The GemStone/J iCommerce Platform. http://www.gemstone.com/products/j/main.html, April 2000.

[89] G.A. Gibson and J. Wilkes. Self-managing network-attached storage. *Communications of the ACM*, 28(4):209–209, December 1996.

[90] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Twenty Third International Conference on Very Large Data Bases*, pages 436–445, 1997.

[91] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, December 1996.

[92] J. Gray and A. Reuter. *Transaction Processing*. Morgan Kaufmann, 1993.

[93] S. Grimstad, D.I.J. Sjøberg, M.P. Atkinson, and R.C. Welland. Evaluating Usability aspects of PJama based on Source Code Measurements. In Morrison et al. [161], pages 307–321.

[94] N. Haines, D. Kindred, J.G. Morrisett, S.M. Nettles, and J.M. Wing. Composing First-Class Transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, November 1994.

[95] C.G. Hamilton. Recovery Management for Sphere: Recovering a Persistent Object Store. Technical Report TR-1999-51, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, December 1999.

[96] C.G. Hamilton, M.P. Atkinson, and M. Dmitriev. Providing Evolution Support for PJama_1 within Sphere. Technical Report TR-1999-50, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, December 1999.

[97] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access With Java: A Tutorial and Annotated Reference*. Addison-Wesley, 1997.

[98] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi Single-Address-Space Operating System. *Software — Practice and Experience*, 28(9):901–928, July 1998.

[99] P.E. Hepp. *A DBS Architecture Supporting Coexisting Query Languages and Data Models*. PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, Scotland, August 1983.

[100] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco*, pages 32–43. Published as SIGPLAN Notices 27(7), July, 1992.

[101] A. Hosking and J. Chen. PM3: An Orthogonal Persistent Systems Programming Language: Design, Implementation and Performance. In Atkinson et al. [23], pages 587–598.

[102] A.L. Hosking, N. Nystrom, Q. Cutts, and K. Brahnmath. Optimizing the read and write barrier for orthogonal persistence. In Morrison et al. [161], pages 149–159.

[103] W.W. Hsu, A.J Smith, and H.C. Young. Analysis of the characteristics of production database workloads and comparison with the TPC benchmarks. Technical Report CSD-99-1070, University of California, Berkeley, November 1999.

[104] R. Hull, R. Morrison, and D. Stemple, editors. *Database Programming Languages*. Morgan Kaufmann, 1989. Proceedings of the Second International Workshop on Database Programming Languages (Salishan Lodge, Gleneden Beach, Oregon, June 1989).

[105] E. Hunt and M.P. Atkinson. Design and Implementation of a Genetics Database using Java and Orthogonal Persistence, June 1998. Poster at the British Biological Sciences Research Council Workshop: Technologies for Functional Genomics, Warwick 1998.

[106] E. Hunt and M.P. Atkinson. Design and Implementation of a Genetics Database using Java and Orthogonal Persistence, August 1998. Poster at Objects in Bioinformatics 1998.

[107] E. Hunt and M.P. Atkinson. PJama: Databases of Indexed Sequence and Mapping Data, April 2000. Poster at Genes, Proteins and Computers Conference, Chester College of Higher Education.

[108] E. Hunt, M.P. Atkinson, R. Irving, I. Darroch, and D. Leader. Visual data exploration and editing using Java, 1999. Poster at the Conference on Datamining in Bioinformatics, Towards *in silico* Biology, Hinxton, Cambridge, 10-12 November.

[109] E. Hunt and D. Jack. Case study: Use of computer tools in locating a human disease gene. Technical report, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, March 1999. TR-1999-28, http://www.dcs.gla.ac.uk/∼ela.

[110] E. Hunt, D. Jack, G.F. Hogg, and D.G. Monckton. Case study: CGT repeat expansion modeling using a Java applet and its PJama extension providing persistent storage for genetics data. Technical report, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, April 1999. TR-1999-31, http://www.dcs.gla.ac.uk/∼ela.

[111] IBM Inc. VisualAge for Java. http://www-4.ibm.com/software/ad/vajava/, 2000.

[112] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In Meissner [150], pages 132–146.

[113] R.P. Japp. Adding Support for Cartographic Generalisation to a Persistent GIS. BSc Dissertation, University of Glasgow, Department of Computing Science, 2000.

[114] D. Jordan. Stimulating Java Persistence — A report on the Second International Workshop on Persistence. *Java Report*, 3(1):26–33, January 1998.

[115] D. Jordan. Serialisation is not a database substitute. *Java Report*, pages 68–79, July 1999.

[116] M.J. Jordan. Early Experiences with PJava. In Jordan and Atkinson [118].

[117] M.J. Jordan. Performance comparisons for a range of persistence strategies and applications, March 2000. Technical report in preperation.

[118] M.J. Jordan and M.P. Atkinson, editors. *Proceedings of the First International Workshop on Persistence and Java.* Number TR-96-58 in Technical Report. Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, November 1996.

[119] M.J. Jordan and M.P. Atkinson, editors. *Proceedings of the Second International Workshop on Persistence and Java.* Number TR-97-63 in Technical Report. Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, December 1997.

[120] M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for Java — A Mid-term Report. In Morrison et al. [161], pages 335–352.

[121] M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for the Java Platform — Specification. Technical report, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 2000. In preparation.

[122] M.J. Jordan and M.L. Van De Vanter. Software Configuration Management in an Object-Oriented Database. In *Proceedings of the Usenix Conference on Object-Oriented Technologies*, Monterey, CA, June 1995.

[123] M.J. Jordan and M.L. Van De Vanter. Modular System Building with Java Packages. In *Proceedings of the Eighth International Conference on Software Engineering Environments*, Cottbus, Germany, May 1997.

[124] P. Kanellakis and J.W. Schmidt, editors. *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann, August 1991.

[125] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Stategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal*, 4(3):519–566, 1995.

[126] G.N.C. Kirby and R. Morrison. OCB: An Object/Class Browser for Java. In Jordan and Atkinson [119], pages 89–105.

[127] G.N.C. Kirby and R. Morrison. Variadic Genericity Through Linguistic Reflection: A Performance Evaluation. In Morrison et al. [161], pages 136–148.

[128] G.N.C. Kirby, R. Morrison, and D.W. Stemple. Linguistic Reflection in Java. *Software — Practice and Experience*, 28(10):1045–1077, 1998.

[129] G.N.C. Kirby, R. Morrison, and D.W. Stemple. Linguistic Reflection in Java: A Quantitative Assessment. In A.L. Brown and C.J. Barter, editors, *Fifth International IDEA Workshop*, Fremantle, Western Australia, 1998.

[130] W. Klas and V. Turau. Persistence in the Object-Oriented Database Programming Language VML. Technical Report TR-92-045, International Computer Science Institute, Berkeley, CA, July 1992.

[131] S. Korsholm. Transparent, Scalable, Efficient OO-Persistence. In *Proceedings of the First ECOOP Workshop on Object-Oriented Databases*, 1999.

[132] K.G. Kulkarni and M.P. Atkinson. EFDM : Extended functional data model. *The Computer Journal*, 29(1):38–45, 1986.

[133] C. Lamb, G. Landis, J. Orestein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[134] Y. Leontiev. *Type System for an Object-Oriented Database Programming Language*. PhD thesis, University of Alberta, Department of Computing Science, 1999.

[135] B. Lewis and B. Mathiske. Efficient Barriers for Persistent Object Caching in a High-Performance Java Virtual Machine. In *Proceedings of the OOPSLA'99 Workshop — Simplicity, Performance and Portability in Virtual Machine Design*, 1999.

[136] B. Lewis, B. Mathiske, and N. Gafter. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine. In Dearle et al. [73].

[137] J. Liedtke. A persistent system in real use: experiences of the first 13 years. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, December 1993.

[138] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[139] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[140] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 230–257. Springer-Verlag, New York, NY, June 1999.

[141] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A.C. Myers. Theta Reference Manual. Technical Report Programming Methodology Group Memo 88, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1995.

[142] D.B. Lomet and M.R. Tuttle. Redo Recovery after System Crashes. In *Proceedings of the Twenty First International Conference on Very Large Data Bases, Zurich, Switzerland*, 1995.

[143] D.B. Lomet and M.R. Tuttle. Logical logging to extend recovery to new domains. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMod-99)*, volume 28(2) of *ACM SIGMOD Record*, pages 73–84, June 1999.

[144] O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, 1993.

[145] B. Mathiske, March 2000. Personal communication.

[146] B. Mathiske and D. Schneider. Automatic Persistent Memory Management for the Spotless Java^{TM} Virtual Machine on the Palm Connected Organizer. Technical Report TR-2000-89, 901 San Antonio Road, MTV-29, Palo Alto, CA 94303, USA, June 2000.

[147] F. Matthes. Higher-Order Persistent Polymorphic Programming in Tycoon. In Atkinson and Welland [25], pages 13–59.

[148] F. Matthes and J.W. Schmidt. Persistent Threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 403–414, Santiago, Chile, September 1994.

[149] M. McAuliffe and M. Solomon. A Trace-Based Simulation of Pointer Swizzling Techniques. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 52–61, Taipei (Taiwan), March 1995.

[150] L. Meissner, editor. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*. Addison-Wesley, Denver, Colorado, USA, November 1999.

[151] P. Milne and K. Walrath. JSR 57 Long-Term Persistence for Java Beans Specification. http://java.-sun.com/aboutJava/communityprocess/jsr/jsr_057_jbprs.html, February 2000.

[152] M.L.B. Mira da Silva. Automating type-safe RPC. In RIDE [182].

[153] M.L.B. Mira da Silva. *Models of Higher-Order, Type-Safe, Distributed Computation over Autonomous Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 1996.

[154] M.L.B. Mira da Silva, M.P. Atkinson, and A. Black. Semantics for Parameter Passing in Type-Complete RPC. In *Proceedings of the International Conference on Data Systems*. IEEE Computer Press, 1996.

[155] C. Mohan. Repeating History beyond ARIES. In Atkinson et al. [23], pages 1–17.

[156] C. Mohan, D. Haderle, B. Lindsay, H. Pirashesh, and P. Schwarz. ARIES : A Transaction Recovery Method supporting Fine-granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[157] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4):378–396, December 1986.

[158] C. Mohan and I. Narang. An efficient and flexible method for archiving a data base. *ACM SIGMOD Record*, 22(2):139–146, June 1993.

[159] R. Morrison, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, D.S. Munro, and M.P. Atkinson. The Napier88 Persistent Programming Language and Environment. In Atkinson and Welland [25], chapter 1.1.3, pages 98–1554.

[160] R. Morrison, R.C.H. Connor, G.N.C. Kirby, and D.S. Munro. Can Java Persist? In Jordan and Atkinson [118].

[161] R. Morrison, M.J. Jordan, and M.P. Atkinson, editors. *Advances in Persistent Object Systems — Proceedings of the Eighth International Workshop on Persistent Object Systems (POS8) and the Third International Workshop on Persistence and Java (PJW3)*. Morgan Kaufmann, August 1998.

[162] Peter Morton. Data Guides for a PJama Database, B.Sc. Dissertation. BSc Dissertation, University of Glasgow, Department of Computing Science, 2000.

[163] J.E.B. Moss. Working With Objects: To Swizzle or Not to Swizzle? Technical Report 90–38, University of Massachusetts, Amherst, Massachusetts, May 1990.

[164] J.E.B. Moss. Working With Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[165] D.S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, Department of Computational Science, University of St Andrews, 1993.

[166] D.S. Munro, R.C.H. Connor, R. Morrison, S. Scheuerl, and D. Stemple. Concurrent shadow paging in the flask architecture. In Atkinson et al. [14], pages 16–42.

[167] R. Munz. SAPNet — the Infrastructure for Business on the web. In Atkinson et al. [23].

[168] N. Nystrom, A.L. Hosking, D. Whitlock, Q.I. Cutts, and A. Diwan. Partial Redundancy Elimination for Access Path Expressions. In *Proceedings of the International Workshop on Aliasing in Object-Oriented Systems (Lisbon, Portugal)*, 1999.

[169] E. Odberg. Category classes: Flexible classification and evolution in object-oriented databases. *Lecture Notes in Computer Science*, 811:406–419, 1994.

[170] H. Ogawa, K. Shimura, S. Matsouka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java. In *Proceedings of ECOOP*, June 2000.

[171] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli — a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD conference*, pages 46–57, Portland, Oregon, 1989.

[172] PJama Team. PJama — API, Tools and Tutorials, web-based documentation, March 2000.

[173] T. Printezis. Analysing a Simple Disk Garbage Collector. In Jordan and Atkinson [118].

[174] T. Printezis. The Sphere User's Guide. Technical Report TR-1999-47, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, July 1999.

[175] T. Printezis. *Management of Long-Running, High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, May 2000.

[176] T. Printezis, M. P. Atkinson, and M. J. Jordan. Defining and Handling Transient Data in PJama. In *Proceedings of the Seventh International Workshop on Database Programming Languages (DBPL'99)*, Kinlochrannoch, Scotland, September 1999.

[177] T. Printezis and M.P. Atkinson. An Efficient Promotion Algorithm for Persistent Object Systems, 2000. To appear in *Software – Practice and Experience*.

[178] T. Printezis, M.P. Atkinson, and L. Daynès. The Implementation of Sphere: a Scalable, Flexible, and Extensible Persistent Object Store. Technical Report TR-1998-46, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, May 1998.

[179] T. Printezis, M.P. Atkinson, L. Daynès, S. Spence, and P. Bailey. The Design of Sphere: a Scalable, Flexible, and Extensible Persistent Object Store. Technical Report TR-1997-45, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, August 1997.

[180] T. Printezis, M.P. Atkinson, L. Daynès, S. Spence, and P.J. Bailey. The Design of a Persistent Object Store for PJama. In Jordan and Atkinson [119], pages 61–74.

[181] J.E. Richardson, M.J. Carey, and D.T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[182] *Proceedings of the Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management (Taipei, Taiwan, March 1995)*. IEEE Computer Society Press, 1995.

[183] J.V.E. Ridgway, C. Thrall, and J.C. Wileden. Towards assessing approaches to Persistence for Java. In Jordan and Atkinson [119], pages 15–36.

[184] J. Rosenberg. The MONADS architecture: A layered view. In *Fourth International Workshop on Persistent Object Systems*, page 207, Martha's Vineyard, MA, September 1990.

[185] J. Rosenberg and D. Koch, editors. *Persistent Object Stores*. Workshops in Computing. Springer-Verlag, 1989.

[186] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[187] C. Russell. JSR-12 Java Data Objects Specification (approved for development). http:/java.sun.com /aboutJava/communityprocess/jsr/jsr_012_dataobj.html, apr 2000.

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

[189] J.W. Schmidt and A.A. Stogny, editors. *Next Generation Information System Technology*. Number 504 in LNCS. Springer-Verlag, 1991.

[190] J.S. Shapiro, J.M. Smith, and D.J. Farber. EROS: a fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 170–185. Kiawah Island Resort, near Charleston, South Carolina, December 1999.

[191] V. Singhal, S.V. Kakkad, and P.R. Wilson. Texas: An Efficient, Portable Persistent Store. In Albano and Morrison [5], pages 11–33.

[192] D.I.K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 1993.

[193] The specjvm98 benchmarks.
http://www.spec.org/osg/jvm98, August 1998.

[194] S. Spence. PJRMI: Remote Method Invocation for Persistent Systems. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99), Edinburgh, Scotland*. IEEE Press, 1999.

[195] S. Spence. *Limited Copies and Leased References for Distributed Persistent Objects*. PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, May 2000.

[196] S. Spence and M.P. Atkinson. A Scalable Model of Distribution Promoting Autonomy of and Cooperation Between PJava Object Stores. In *Proceedings of the 13th Hawaii International Conference on System Sciences*, Hawaii, USA, January 1997.

[197] M. Stonebraker. *Readings in Database Systems*. Morgan Kaufmann, 1988.

[198] M. Stonebraker and D. Moore. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann, San Francisco, 1996.

[199] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatso, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[200] Sun Microsystems, Inc. Enterprise Java Beans Specification 1.1, April 2000.

[201] Sun Microsystems, Inc. Forte For Java. http://www.sun.com/forte/ffj/, 2000.

[202] Sun Microsystems, Inc. Java Blend. http://www.sun.com/software/javablend/index.html, April 2000.

[203] Sun Microsystems, Inc. K Virtual Machine (KVM). http://java.sun.com/products/kvm/, 2000.

[204] Sun Microsystems, Inc. The Sun Ray1 Enterprise Appliance Technical Documentation, April 2000.

[205] A. Taivalsaari. Implementing a Java Virtual Machine in the Java Programming Language. Technical Report TR-98-64, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 1998.

[206] A. Taivalsaari, B. Bush, and D. Simon. The Spotless System: Implementing a Java System for the Palm Pilot Connected Organizer. Technical Report TR-99-73, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 1999.

[207] S.M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In Dittrich and Dayal [75], pages 148–159.

[208] C.A. Thekkath, J. Wilkes, and E.D. Lazowska. Techniques for file system simulation. *Software — Practice and Experience*, 24(11):981–999, November 1994.

[209] Tichy. Should computer scientists experiment more? *COMPUTER: IEEE Computer*, 31:32–40, 1998.

[210] United Kingdom Ordnance Survey. *Strategi User Guide*, 1997. http://www.o-s.co.uk/downloads/vector/strategi/Strategi_w.pdf.

[211] US Census Bureau. Tiger line files, technical documentation. http://www.census.gov/ftp/pub /geo/www/tiger/, 1998.

[212] M.L. Van De Vanter. Coordinated Editing of Versioned Packages in the JP Programming Environment. In *Proceedings of the Eighth International Symposium on System Configuration Management (SCM-8) Brussels*, LNCS. Springer-Verlag, 1998.

[213] M.L. Van De Vanter and T. Murer. Global Names: Support for Managing Software in a World of Virtual Organizations. In *Proceedings of the Nineth International Symposium on System Configuration Management (SCM-9) Toulouse, France*, LNCS. Springer-Verlag, 1999.

[214] W3C.com. Jigsaw — the w3c's web server. http://www.w3.org/Jigsaw/, 2000.

[215] F. Wai. Distributed PS-algol. In Rosenberg and Koch [185], pages 126–140.

[216] P. Wegner and D. Goldin. Interaction as a framework for modeling. *LNCS*, 1565:243–257, 1999.

[217] P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *ECOOP '88, European Conference on Object-Oriented Programming, Oslo, Norway*, volume 322 of *LNCS*, pages 55–77. Springer-Verlag, August 1988.

[218] R.C. Welland and M.P. Atkinson. A Zoned Architecture for Large-Scale System Evolution. In *Proceedings of the Third International Software Architecture Workshop*, pages 155–158. ACM, November 1998.

[219] D. White and A. Garthwaite. The GC interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 1998.

[220] S.J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin, Madison, Madison, WI, September 1994.

[221] S.J. White and D.J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *International Conference On Very Large Data Bases (VLDB '92)*, pages 419–431. Morgan Kaufmann, August 1992.

[222] D. Whitlock and A. Hosking. A framework for persistence-enabled optimization of Java applications. In Dearle et al. [73].

[223] J. Wilkes, R.A. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computers and Systems*, 14(1):108–136, February 1996.

[224] P.R. Wilson and V.B. Balayoghan. Compressed paging. Personal communication, 1995.

[225] P.R. Wilson and S.V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *Proceedings of the 1992 Workshop on Object-Orientation in Operating Systems (September 1992, Dourdan, France)*, pages 364–377. IEEE Computer Society Press, 1992.

[226] W.A. Woods. Conceptual Indexing: A Better Way to Organize Knowledge. Technical Report SMLI TR 97-61, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 1997.

[227] W.A. Woods, L.A. Bookman, A. Houston, R.J. Kuhns, P. Martin, and S. Green. Linguistic Knowledge can Improve Information Retrieval. Technical Report SMLI TR 99-83, Sun Microsystems Laboratories, 901 San Antonio Road, Palo Alto, CA 94303, USA, 1999.

[228] S.B. Zdonik. Version management in an object-oriented database. In *Proceedings of the IFIP International Workshop on Advanced Programming Environments*, pages 405–422, Trondheim, Norway, June 1987.

[229] R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The story of* O_2. Morgan Kaufmann, 1992.

[230] E. Zirintsis, V.S. Dunstan, G.N.C. Kirby, and R. Morrison. Hyper-Programming in Java. In Morrison et al. [161], pages 370–382.

About the Authors

Malcolm Atkinson is a Visiting Professor at Sun Microsystems Laboratories, and has been a full Professor at the University of Glasgow, Scotland since 1984. He has sought to improve the context for the construction of large and complex applications via the provision of better integration between programming languages and databases, since working in Neil Wiseman's Rainbow group on CAD and Graphics with Mick Jordan in the early 1970s. He identified the value of orthogonal persistence at VLDB in 1978 and led the team that built the first orthogonally persistent programming language, PS-algol, in 1980. He currently leads research projects in bioinformatics, cultural computing, interpretation of remote-observations of users' actions, computer support for distance learning and persistence. He is a Fellow of the British Computer Society and a Fellow of the Royal Society of Edinburgh. He received his Ph.D. from the University of Cambridge, England in 1974.

Mick Jordan is currently a Senior staff Engineer and Principal Investigator at Sun Microsystems Laboratories. His interests include programming languages, programming environments, software configuration management, and persistent object systems. He has a Ph.D. in Computer Science from the University of Cambridge, UK.