UNIVERSITY
of
GLASGOW

Department of
Computing Science

Ph.D. Thesis

Management of Long-Running High-Performance Persistent Object Stores

Antonios Printezis

Submitted for the degree of

Doctor of Philosophy

at the University of Glasgow

May 2000

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

This thesis was typeset by the author, using *exclusively* the LaTeX 2ε document preparation system, in conjunction with Emacs for editing, BibTeX for the bibliography, Xfig for all diagrams, Gnuplot for all graphs, and CVS for version control. These were run on an Sun Ultra 2 Creator running Sun Solaris 7 and an IBM ThinkPad 380ED running Linux. The fonts used were Times for normal text, AvantGarde for the chapter headings, Helvetica for the section headings, page numbers, and some code fragments, `Courier` for further code fragments, and *ZapfChancery* for the quotations. All the above software can be obtained totally free of charge (apart from Solaris, which can be obtained free of charge for personal use only).

Alright, I'll give it a try.
*— **Luke Skywalker**, Jedi Apprentice*
No! Try not. Do... or do not.
*— **Yoda**, Jedi Master*

Thesis Statement

There is a growing demand for large persistent object stores that can be used for long periods and by diverse applications.

Work on an orthogonally persistent platform for Java revealed the need for a scalable persistent object store, capable of continuous operation and schema evolution, while supporting a multi-threaded transactional load and offering recovery after failures. Recognising the high cost of building a store of this capability, a generic approach capable of supporting a wide variety of loads was chosen.

It was believed that the appropriate trade-offs between properties and flexibility could best be achieved by splitting the store into two parts: the store core and the application-specific part. By keeping the core simple and including the required level of complexity in the application-specific part, the right balance between resource requirements, complexity, flexibility, and performance is achieved. This approach is also ideal for research purposes, as it encourages modification, extension, and experimentation.

The feasibility and benefits of the above architecture have been demonstrated as it was adopted in the Sphere persistent object store. In Sphere even fundamental facilities, such as bulk object-loading, garbage collection, and schema evolution, are in the application-specific part of the store. This allows them to be specifically implemented and optimised for the load that Sphere has to support.

Sphere is currently integrated in $PJama_1$ and used by a number of substantial applications.

You refer to the prophecy of The One, who will bring balance to the Force.
You believe it's this boy?
*— **Mace Windu**, Jedi Master*

Abstract

The popularity of object-oriented programming languages, such as Java and C++, for large application development has stirred an interest in improved technologies for high-performance, reliable, and scalable object storage. Such storage systems are typically referred to as *Persistent Object Stores*.

This thesis describes the design and implementation of Sphere, a new persistent object store developed at the University of Glasgow, Scotland. The requirements for Sphere included high performance, support for transactional multi-threaded loads, scalability, extensibility, portability, reliability, referential integrity via the use of disk garbage collection, provision for flexible schema evolution, and minimised interaction with the mutator.

The Sphere architecture is split into two parts: the core and the application-specific customisations. The core was designed to be modular, in order to encourage research and experimentation, and to be as light-weight as possible, in an attempt to achieve high performance through simplicity. The customisation part includes the code that deals with and is optimised for the specific load of the application that Sphere has to support: object formats, free-space management, etc. Even though specialising this part of the store is not trivial, it has the benefit that the interaction between the mutator and Sphere is direct and more efficient, as translation layers are not necessary.

Major design decisions for Sphere included *(i)* splitting the store into partitions, to facilitate incremental disk garbage collection and schema evolution, *(ii)* using a flexible two-level free-space management, *(iii)* introducing a three-dimensional method-dispatch matrix to invoke store operations, which contributes to Sphere's ease-of-extensibility, *(iv)* adopting a logical addressing scheme, to allow straightforward object and partition relocation, *(v)* requiring that Sphere can identify reference fields inside objects, so that it does not have to interact with the mutator in order to do so, and *(vi)* adopting the well-known ARIES recovery algorithm to ensure fault-tolerance.

The thesis contains a detailed overview of Sphere and the context in which it was developed. Then, it concentrates on two areas that were explored using Sphere as the implementation platform. First, bulk object-loading issues are discussed and the *Ghosted Allocation* promotion algorithm is described. This algorithm was designed to allocate large numbers of objects to a store efficiently and with minimal log traffic and was evaluated using large-scale experiments. Second, the disk garbage collection framework of Sphere is overviewed and the implemented compacting, relocating garbage collector is described, along with the model of synchronisation with the mutator.

Your overconfidence is your weakness.
*— **Luke Skywalker**, Jedi Knight*
Your faith in your friends is yours.
*— **Emperor Palpatine**, Ruler of the Empire*

Acknowledgements

I am grateful to the following people for their endless support, constant encouragement, and constructive contributions that made this thesis possible. *May the Force be with you. Always.*

- *Malcolm Atkinson*, my never-tiring Ph.D. supervisor, for his outstanding enthusiasm and never-ending energy, the constant flow of ideas he generated, his heroic attempts to read drafts of chapters of this thesis by not getting any sleep, and most importantly for the single most useful piece of advice I've ever been given:

  "Coffee never tastes as good if it doesn't have some old coffee in it."

- *Quintin Cutts*, for undertaking the difficult task of being my second supervisor.

- *Peter Dickman*, for the constructive garbage collection-related discussions and for providing the initial ideas for the algorithm described in section 4.7.3.

- *Craig Hamilton*, who single-handedly developed the logging and recovery sub-systems of Sphere and provided extra facilities, promptly and reliably, that made possible the experiments presented throughout this thesis.

- My other two office mates, *Susan Spence* and *Huw Evans*, for patiently correcting my written English, as *"I speak English no good"*, sharing my enthusiasm for good food, gourmet cooking, and malt whisky, or, as was most often the case, combining all the above activities in various pubs around Glasgow and other parts of the world.

- *Laurent Daynès*, for his considerable contributions to the initial Sphere design and for constantly expanding my French vocabulary.

- The original members of the Forest group at SunLabs West, *Mick Jordan* and *Michael Van De Vanter*, for an enjoyable internship in California during the summer of 1996, and the rest of group, *Brian Lewis*, *Bernd Mathiske*, and *Neal Gafter*, for the interesting and passionate debates over the past three years.

- The members of the Java Topics (now Java Technology Research) group at SunLabs East, *Steve Heller*, *Ole Agesen*, *Dave Detlefs*, *Christine Flood*, *Alex Garthwaite*, *Guy Steele*, and *Derek White*,

as well as *Ross Knippel* from SunSoft in CA, for a constructive internship in Massachusets during the summer of 1998. Working in a product group under extreme pressure of deadlines was an amazing experience for me and increased considerably my application developing, debugging, and testing skills. Getting a boiler-suit for my trouble too was an unexpected but welcome bonus.

- *Tony Hosking* and *Paul Cockshott*, my external and internal examiner respectively, for their useful and constructive comments on this thesis.

- My mum and dad, *Mina Printezi* and *George Printezis*, for bringing me up, supporting me, and putting up with my anti-social behaviour during my years at University.

- Finally, a special mention should go to *Mr Costas Stathopoulos* for introducing me to Scotland. If it was not for him, I would probably still think that Glasgow is a suburb of London.

- The research presented in this thesis was supported by a Glasgow University postgraduate scholarship that covered both fees and living expenses for three years, a collaborative research grant from Sun Microsystems Inc., and grant number GR/K87791 from the British Engineering and Physical Science Research Council (EPSRC). The machine used to run all the experiments presented throughout this thesis was kindly donated by Sun Microsystems Inc.

Antonios Printezis, May 2000

At last we will reveal ourselves to the Jedi. At last we will have revenge.
*— **Darth Maul**, Dark Lord of the Sith*

Contents

An analysis of the plans provided by Princess Leia has demonstrated a weakness in the battle station.
*— **Jan Dodonna**, Rebel Alliance General*

List of Figures

I have placed information vital to the survival of the Rebellion into the memory systems of this R2 unit.
— ***Leia Organa**, Princess of Alderaan*

List of Tables

Impressive. Most impressive. Obi-Wan has taught you well. You have controlled your fear. Now, release your anger! Only your hatred can destroy me!
— **Darth Vader**, *Dark Lord of the Sith*

Chapter 1

Introduction

The demand for bigger, better, more scalable, and more reliable applications has existed since the early days of information technology. Such applications have been referred to in the past as *Persistent Application Systems* [AM95]. However the current "buzzword" for them is *Enterprise Applications*. This demand is still constantly growing, encouraged by rapid advances in technology (CPU and memory speeds, hard disk capacity and access times, etc.). The associated decrease in hardware costs have also allowed this technology to be more widely available than ever before.

The vast majority of enterprise applications require a combination of sophisticated data models, complex computation, and persistence [AJ00]. The source of these requirements is the human ability to comprehend complex problems, establish mental models, and accomplish tasks, individually or collaboratively, over long periods of time. Lately, the complexity of these tasks has increased dramatically, due to worldwide sharing of data and resources through high-speed networks.

In order to develop such large applications, appropriate tools are needed. At present a combination of database systems, programming languages, and software development environments are widely used in this role. Due to the typically heterogeneous nature of such tools, the development and maintenance of large applications still remains an unnecessarily challenging, time-consuming, and costly task. Hence, any radical improvements in this process are not only interesting research topics, but can also be of serious commercial significance.

Stonebraker *et al.* observe that *"many applications are on a path of increasing complexity towards sophisticated data models and complex logic and persistence"* and that *"current solutions tend to address only the simple business applications"* [SBM96]. In an effort to achieve a homogeneous and integrated solution to software development, *Orthogonal Persistence* was proposed by Atkinson in 1978 [Atk78] (it is further discussed in section 2.2). It provides seamless integration between the programming language and the storage mechanism that renders unnecessary the need for a separate database system, along with the complexity such separation introduces: a schema modelling language, restrictions imposed on the development of the application, the need for developers to be familiarised with it, etc. Instead, in *Orthogonally Persistent Systems*, operations such as data access, data modelling, and ensuring schema consistency are achieved through

a way that is familiar and natural to the developer: the constructs of the programming language.

The storage technology needed to support such systems is required to reliably accommodate data in a format similar to the one used by the programming language. This requirement is fundamental for the transformation-free migration of data between main-memory and persistent storage, which increases efficiency and decreases complexity.

1.1 Persistent Object Stores

For almost the past thirty years, by far the most popular and widely-used technology for storing large amounts of data has been *Relational Databases* (RDBs) [EN94]. Such systems use the *Relational Model*, first proposed by Codd in 1970 [Cod70], to store, organise, and retrieve data. According to this model, the data is organised in tables, also referred to as *Relations*, with all of the data on each column being of the same data type. Queries against the data consist of retrieving one or more rows of a table, according to given constraints. Facilities for merging tables, selecting specific columns, using indexes for faster look-ups, etc. are also provided. Users can query the data using special-purpose query languages, by far the most popular of which is the *Simple Query Language* (SQL) [MS93, EN94].

Relational systems gained popularity because their storage and query model is simple and easy to understand. Further, some facilities they provide aid considerably the development, evolution, and maintenance of large, long-running applications. Such facilities include a straightforward way of evolving relations (addition/removal of columns and other tables), dynamic binding to code (all RDBs support parsing and evaluation of queries at runtime), logical stability through the use of views (producing the same consistent view of the data, irrespective of changes to the tables and their logic), etc. However, the main reason behind their current wide use more pragmatic. Large corporations (and individual users for that matter!) will only store important and mission-critical data on systems that they trust and that have been operationally proven. Relational systems, which have existed for decades and have been widely deployed, provide exactly this: a mature, robust, trusted, and well-proven technology for storing data.

However, during the last decade or so, there has been an increasing interest in databases that store large quantities of very complex and highly-connected data. Examples of such data are R-trees [Gut84], used in *Geographical Information Systems* (GIS) to query maps, DNA sequences [Dun99], complex 3D models [FvFH93], etc. It turns out that, even though traditional RDBs are good for storing and querying data that can inherently be expressed as tables (telephone directories, employee salary information, etc.), they are not very appropriate for storing the complex data mentioned above. There are two reasons for this.

- The schemata necessary to store such data are very complicated, hard to maintain, and obscure the actual algorithms that need to operate over the data.
- Most importantly, accessing such data can be very inefficient, as some of the required queries cannot be expressed in the traditional relational calculus. This can be solved by the use of multiple queries, possibly compromising efficiency and clarity.

The two points above contradict the claims of simplicity and efficiency of RDBs. This has stirred research in alternative database systems that can store such data more naturally and efficiently. Such systems are called *Persistent Object Stores* (POSs) or *Object-Oriented Databases* (OODBs). The difference between these two terminologies is that POS typically refers to the low-level storage technology, whereas OODB refers to a

complete database system, comprising query facilities, language mappings, etc. [ABD+92].

The unit of storage of a POS is an *Object*, which is defined in section 3.4.1, in contrast to the table, in the case of an RDB. Since there is a unique identifier associated with every object stored in a POS, the notion of a *Persistent Reference* is introduced. Using such references, stored objects can point to others (this is similar to memory pointers in programming languages like C [KR88] and Pascal [FW85] or object references in languages like Java [GJS96]). The combination of objects and references allows the creation of large and complex object graphs, where each node is represented by an object and each edge by a reference. In fact, such object graphs can mirror in-memory data structures and can be manipulated as such by programmers (as is the case for PJama, ⇝§2). Object graphs can also represent a wide range of data that are inherently difficult to express in terms of relations. Examples of such data are inherently recursive data structures (e.g. a wide variety of trees), network structures (road networks, electrical circuits, chip designs, flow graphs, etc.), scientific and engineering data (e.g. environmental statistics, fish distributions, airframe designs, etc.) and so on.

Some interesting issues that arise when constructing a POS are enumerated below.

- **Fault-Tolerance** — POSs include fault-tolerance facilities to protect the stored data from crashes, power failures, etc. Typically, any updates to the objects in a POS take place in terms of a *transaction* [GR93]. All updates that belong to the same transaction are *atomically* written to the store; i.e. they are written in their entirety or not at all. This ensures that the store is always accessed in a consistent state. Usually, either *logging* [MHL+89, MHL+92] or *shadow-paging* [MCM+94] techniques are used to ensure atomicity.

- **Size** — The size of POSs vary considerably according to: *(i)* the context in which they are used, *(ii)* the overhead of any management data structures, and *(iii)* limitations of the implementation. Even though relatively small POSs, in the order of a few megabytes, are common (e.g. persistent facilities for organisers, laptops, etc.), much larger ones, in the order of gigabytes, have also been reported in commercial environments [Oti98]. Because the price of disk storage is steadily decreasing, the latter case is bound to become more common.

- **Concurrency Control** — Most non-trivial applications require concurrent access to the persistent storage, whether this is in the form of multiple (possibly remote) clients accessing the same server (*client-server* model) or in the form of multiple threads running against the same store (*single-user* model). In either case, concurrency control must be in place in order to avoid any data corruption caused by the concurrent access.

- **Free-Space Management** — During the lifetime of a POS, objects are allocated in it, fetched from it, and their contents are updated. Additionally, objects also need to be discarded when they are not used any more (i.e. when all references to them have been deleted and the objects become *garbage* [Jon96]). There are two models of discarding objects: *(i)* by explicit de-allocation requests (as in ObjectStore [LLOW91], Texas [SKW92, WK92], and Shore [CDF+94]) or *(ii)* by automatic free-space reclamation techniques (i.e. *garbage collection* [Jon96], as in Thor [LAC+96], Larchant [SF95, FS95], and GemStone/J [Gem98b, Gem98a]). The free-space management mechanism of a POS should depend on and be optimised for the approach taken.

- **Schema Evolution** — Long-lived applications need to evolve over time, as developers fix bugs, introduce improvements, or deal with changes to the requirements. Such evolution not only involves changes to object contents but also changes to the object types and hence layout. It is necessary for

POSs to provide a mechanism for the objects they contain to be evolved in this way. This is referred to as *Schema Evolution* [DA99b]. Lack of such a facility severely limits the longevity and hence usefulness of a POS, as changes in the implementation of an application might require store repopulation. This is not always feasible as the original data might not be available any more.

Considering the above points, a POS can be seen as a very large non-volatile dynamically-allocating heap. In fact, in the case of persistent programming languages (PJama ⇝§2, Napier88 [MCC+99], Tycoon [Mat99], etc.), this is exactly the programming model presented to the application developer.

There is evidence that, for many situations, it is beneficial to use a POS rather than an RDB. In particular, performance gains [DAV97, RTW97] and maintainability and ease-of-development improvements [OLC97] have been reported. Unfortunately, despite their advantages, OODBs/POSs are unlikely to replace RDBs in the near future. The main reason behind this is the trustworthiness of the RDBs, as discussed earlier in this section, and the huge amounts of legacy data that already reside on RDBs. Currently, it is believed that *"OODBs/POSs will remain a niche market"* [Mun99]. However, since this is mainly due to political rather than fundamental reasons, it is still possible that OODBs/POSs will gain trustworthiness and popularity through heavy-duty deployments, as RDBs did twenty years ago.

1.2 Overview of Persistent Languages and Object Stores

There have been a large number of systems developed to provide persistence for objects. The most notable of these, but by no means all of them, are presented in this section.

The first system to adopt orthogonal persistence was the pioneering *PS-algol* language and system [ACC82, ACC83a] from the Universities of Edinburgh and St Andrews, Scotland. Among other important innovations, it introduced the technique of *pointer-swizzling*, i.e. overwriting persistent identifiers in main-memory cached objects with memory addresses to optimise subsequent object accesses.

PS-algol was succeeded by *Napier88*, which employed a more powerful type system and other implementation improvements [MCC+99, CCM99]. Other similar offerings were *Tycoon* [Mat99, MSS99] from the University of Hamburg, Germany, *Fibonacci* [AGO99] from the University of Pisa, Italy, and lately *PJama* (⇝§2) from the University of Glasgow, Scotland and Sun Microsystems Research Laboratories West, California. All these systems adopted transparent orthogonal persistence at the object-level (more information on the history of these systems is given in section 2.2.5).

The Napier88 system used a purpose-built POS, usually referred to as the *Napier Store* [Bro89, BM92, BR91, BMM+99]. It was designed to be generic and was actually used by other systems, one of them being an implementation of Tycoon [MMS99]. A derivative of the Napier store was *Flask* [KCC+97, Mun93], which introduced concurrency and distribution. Both these stores used *shadow paging* for their recovery facilities [MCM+94].

There are many other systems that do not embrace orthogonal persistence but rely on different approaches to identify which objects should persist. The vast majority of them are targetted for C++ [ISO98] or language-extensions to C++. *Avalon/C++* [EMS91] from Carnegie Mellon University (CMU) is a language designed to support reliable distributed computing. It uses the inheritance mechanism of C++ to provide properties to objects, like synchronisation and recovery. The associated storage mechanism is called *Camelot* [EMS91].

The *EXODUS* storage manager [CDRS86, CD87] from the University of Wisconsin—Madison provides a client-server architecture with log-based recovery [FZT+92] using the ARIES algorithm [MHL+92] and support for multiple servers and distributed transactions. Its objects are untyped sequences of bytes. EXODUS is distributed with the *E* programming language [RCS93], a variant of C++ that supports creation and manipulation of persistent data structures. Its creators claim that E is the first C++ variant to do so transparently. EXODUS was succeeded by *SHORE* [CDF+94], which introduced typed objects (the user defines the type of objects using the *SHORE Data Language* or *SDL*), access control, indexing and clustering facilities, and other improvements.

The *ObjectStore* system [LLOW91] from ObjectDesign Inc.[1] is one of the most successful commercial object-oriented database products and is widely used in data-intensive applications such as CAD, CASE, GIS, etc. Again it targets C++ and persistence is provided by inheritance from a special storage class or alternatively by allocating objects within a persistent collection. ObjectStore has also adopted a client-server model. However, if both the client and the server are running on the same machine, communication between them can be performed via shared memory in an attempt to increase performance. Lately, ObjectDesign Inc. announced new products targetting the Java language (⇝§2.7).

The *Texas* store [SKW92, Kak98] from the University of Texas at Austin also targets C++ but does not rely on a storage class to provide persistence facilities to objects. Instead, an object is designated as persistent at allocation-time. Texas introduced the *pointer-swizzling at page-fault time* technique [WK92] that overwrites all pointer-fields on a page with memory addresses when the page is faulted-in and reserves address space but not physical memory for pages that have not yet been fetched. The pointer locations in objects are discovered using Texas' portable runtime type descriptors [KJW98].

Moving to more general systems, O_2 [BDK92] is an object-oriented database developed by the Altaïr Consortium (later O_2 Technology). It comprises an object storage system, language bindings (e.g. BasicO_2, CO_2), a set of user-interface generation tools, and a programming environment. O_2 was a pioneering system as it was the first full-fledged object-oriented database. Another important contribution of this project was that it introduced the *Object Query Language* (OQL) [ASL89], which is similar to SQL [MS93] but was designed to query collections of objects.

Thor [LAC+96] is an object-oriented database for distributed systems from the Massachusets Institute of Technology (MIT). It supports heterogeneity at several levels: machine, network, operating system, and programming language. It provides high reliability and availability so that persistent objects are likely to be accessible despite failures. Thor supports *type-safe-sharing* of persistent objects. Safety is ensured by the fact that code can only access objects via their methods. Objects are stored along with their methods, which are implemented in the *Theta* programming language [LCD+94]. Client code can be written in unsafe languages, such as C++, however it is executed in separate protection domains. In this manner, Thor guarantees the integrity of the store. Theta and Thor also provide automated memory management to avoid dangling pointers and space leaks [ML97].

Larchant [SF95, FS96] from INRIA is a distributed shared memory that can persist on reliable storage. Memory management in Larchant is automatic, reclaiming unreachable objects and identifying reachable objects from persistent roots in order to write them to disk. Its main contribution is the novel garbage collection algorithm that it employs, which minimises I/O and locking traffic and requires no synchronisation between the garbage collector and application (the collector is described in more detail in section 6.9).

[1] ObjectDesign Inc. has been recently renamed to eXcelon Corp.

Larchant has been mainly targetted for uncooperative[2] programming languages (e.g. C++), but the implementers claim that it should work equally well with cooperative environments (e.g. Java).

Finally, *Mneme* [Mos90a] is a POS from the University of Massachusets at Amherst (UMass). It was built in the context of the Mneme project, whose goal was the implementation of efficient, distributed, and persistent programming languages. Design requirements for the Mneme store included portability, high performance, and extensibility with respect to object management. The memory model of the store aimed to present the programmer with a single, shared heap, distributed across a number of computers. Mneme was used as a basis for the Persistent Smalltalk system [HMB90], as well as information retrieval systems.

In addition to the systems presented here, a considerable number of persistence solutions have been developed for the Java programming language [GJS96]. These are overviewed in section 2.7.

1.3 Motivation

This thesis describes Sphere, a new persistent object store developed at the Department of Computing Science of the University of Glasgow in Scotland. It was developed in the context of the PJama project, a collaborative effort between the University of Glasgow and Sun Microsystems Research Laboratories in California to provide orthogonal persistence for the Java programming language (⇝§2). The main design goals behind the development of Sphere were high performance, support for transactional multi-threaded loads, scalability, extensibility, portability, reliability, store integrity via the use of disk garbage collection, provision for flexible schema evolution, and minimised interaction with the mutator (these are discussed further in section 3.2).

When a new POS was sought to support the PJama system, using an existing one, rather than developing a new one, would have been easier and less risky. However, none of the existing technologies provided the correct balance between performance, flexibility, and light-weightiness that the PJama system required. In particular,

- when light-weight, they did not provide the flexibility required by PJama, i.e. they imposed fixed object formats (e.g. Napier store), adopted an explicit de-allocation scheme (e.g. Texas, ObjectStore), typically along with a physical addressing model (e.g. Napier store, Texas), etc., or
- when enough flexibility and facilities were provided, the store was relatively heavy-weight, as it usually assumed a client-server environment (e.g. EXODUS, SHORE, ObjectStore, O_2, Thor) that PJama did not need.

Additionally, one of the goals of the PJama project was to pursue research into very large-scale persistence technologies for object storage, especially in the areas of disk garbage collection (⇝§6), bulk object-loading (⇝§5), schema evolution (⇝§6.7), size scalability, and longevity. Because of this, it was necessary that the adopted POS could be easily modified and extended so that new algorithms are implemented, tuned, and compared. The PJama team members felt that only a locally-developed system could provide the appropriate interfaces and abstractions that could make this possible. Finally, the requirement of being able to modify and instrument the code eliminated the use of any commercial products (e.g. ObjectStore, O_2), as source code for them is typically not supplied.

[2] An uncooperative runtime environment is one that cannot accurately determine the exact location of memory reference in its runtime stacks and/or memory heap.

1.4 Thesis Overview

The thesis is organised as follows.

Chapter 2 overviews the PJama project. It describes its goals, the programming model it has adopted, the issues that were raised during its development, its two implementations, and establishes the context in which Sphere was developed.

The design of Sphere is given in chapter 3. It covers its original requirements, the abstractions introduced in Sphere, the high-level management of the store, and free-space-management issues.

Chapter 4 gives details on the implementation of Sphere. It overviews the main Sphere components, presents how they were implemented, how they interact with each other, and any optimisations that were adopted to improve performance. It also presents measurements for some of the components to illustrate how they operate.

Bulk object-loading issues are covered in chapter 5. This is where *Ghosted Allocation*, Sphere's object promotion algorithm, is presented along with the results of large-scale experiments that evaluated its performance and tuned several parameters of its implementation.

Chapter 6 overviews the garbage collection framework of Sphere. It establishes the need for garbage collection in POSs, describes the operation of the Sphere compacting garbage collector, its synchronisation with mutator threads, and how it was adopted to also be used as the schema evolution mechanism. An evaluation of the first prototype of the Sphere garbage collector is also included.

Results from evaluation experiments are included in chapter 7. They illustrate the performance and ease-of-use benefits of using PJama and Sphere over two alternative persistence mechanisms.

Finally, chapter 8 summarises the thesis and presents areas of future work.

Don't be too proud of this technological terror you've constructed. The ability to destroy a planet is insignificant next to the power of the Force.
— **Darth Vader**, *Dark Lord of the Sith*

Chapter 2

Overview of the PJama System

This chapter gives a brief overview of the PJama system. Section 2.1 describes the Java programming language and section 2.2 introduces the concepts of persistence and orthogonal persistence. Section 2.3 provides an overview of the goals of the PJama project. Sections 2.4 and 2.5 describe the architecture of PJama Classic, the first PJama prototype, and PEVM, the subsequent high-performance PJama implementation, respectively. Section 2.6 summarises the future directions of the project and section 2.7 overviews other persistence solutions for Java. Section 2.8 concludes the chapter.

2.1 The Java Programming Language

Late in 1990, Sun Microsystems Inc. started a product-focused project called *Green*. James Gosling, one of its original members, describes its goals as follows.

> *"[The Green Project] was chartered to spend time (and money!) trying to figure out what would be the 'next wave' of computing and how we might catch it. We quickly came to the conclusion that at least one of the waves was going to be the convergence of digitally controlled consumer devices and computers."*

So, the Green project was focused on consumer devices, like televisions, videos, laser-disc players, and stereos, and how they can communicate with each other. Such devices are typically made with different CPUs and have limited amounts of program memory. This encouraged the team to develop a new programming language, designed to allow programmers to support more easily dynamic, changeable hardware. Gosling named this language Oak, after a large oak tree that resided outside his window.

Oak [Gos93] was an object-oriented programming language, syntactically very similar to C++ [ISO98]. However, it was "stripped-down" to a bare minimum to be compatible with the limited space small devices would offer. It supported single inheritance through implementation and multiple inheritance through interfaces. It was also type-safe and its memory management relied on a garbage collector [Jon96]. The combination of these two features eliminated the dangling-reference problems that plague C/C++ programs. Some extra features included in Oak were exceptions, built-in concurrency facilities (e.g. thread synchronisation),

and assertions (even though the latter were eventually dropped). Finally, in order to be totally portable, Oak was not designed to be fully-compiled. Instead, it was compiled to an intermediate architecture-independent bytecode format, that was in turn executed by the Oak bytecode interpreter.

Ultimately, the Green project never achieved any commercial success and was eventually cancelled, even though a prototype interactive remote control, called *Star7*, was successfully built and demonstrated. Around that time, in 1993, the *National Center for Supercomputing Applications* (NCSA) introduced the *World Wide Web* (WWW) and the Mosaic (later Netscape) browser. These provided computer users with a revolutionary uniform way of accessing information, regardless of the machine architecture they used. It was clear then that the architecture-neutral qualities of the Oak language could allow it to be used in this context and add dynamic content to the otherwise static information provided by HTML, the text-based mark-up language used for information transfer in the WWW [RLJ98]. So in 1994 Oak was retargetted to the WWW and was renamed *Java*. Eventually, a deal was made with Netscape Corp. for Java to be included in the Netscape browser.

Only five years later, Java is hailed as one of the biggest advances in computing[1]. Software companies have started marketing products entirely written in Java and developers report improvements by up to a factor of two in development time and by up to a factor of three in robustness, when using Java instead of C++ [Phi99]. The Java language has been augmented with a large set of standard classes and APIs, covering a large number of facilities needed by developers, like 2D/3D graphics (Java 2D/3D APIs [Sun99b, Sun99a]), networking (Java RMI [Sun98d]), relational database access (JDBC [Sun98e]), component technology (JavaBeans [Sun97]), etc. The combination of the Java language and the set of standard APIs (currently comprising several thousand Java classes and interfaces) is known as the *Java Platform*. The existence of these APIs ensures the portability of the code when moved to different platforms. A number of companies are marketing complete Java platform systems and development environments: Visual Café from Symantec, JBuilder from Borland, J++ from Microsoft, etc. However, the original *Java Development Kit* (JDK) from Sun Microsystems Inc., currently in version 1.2 [Sun98c], is considered to be the standard[2].

The official definition of the Java language is given in the *Java Language Specification* (JLS) by Gosling, Joy, and Steele [GJS96]. The official definition of the Java Virtual Machine Specification is given by Lindholm and Yellin [LY99]. More information on the history of Java and the Green project is given by Gosling [Goswww], Byous [Byowww], and English [Engwww].

2.2 Orthogonal Persistence

All memory-resident data, which is created and manipulated during the execution of a given process, is typically automatically discarded by the operating system at the end of its execution[3]. However, most non-trivial applications need to retain at least some data across multiple invocations. This concept is called *Persistence*. It is usually achieved by storing the data on secondary storage with the use of ad-hoc custom-made schemes, programming language facilities (e.g. pickling [BNOW93] in Modula3 [CDG+89] or object

[1] Ironically enough, Java did not introduce any new concepts or ideas. Instead, it incorporated a lot of already-existing ones, like garbage collection and bytecode-interpretation. It was Java's success that ultimately made them commercially-acceptable.

[2] Lately, JDK has been renamed to *Java™ Software Development Kit* (SDK). However, JDK will be used instead throughout this thesis.

[3] This is not entirely true, in the case of persistent operating systems like Grasshopper [DdF+94] and L4 [Lie96], in which processes, including their address spaces, can be made persistent and survive across machine power-downs. Such systems however are still at an experimental stage and are not widely used.

serialisation [Sun98b] in Java [GJS96]), or heavy-duty databases (e.g. relational [EN94, Cod70] or object-oriented [ABD+92, Cat97, Coo97]). All the above systems require some effort from the programmer in order to operate correctly (mapping program data structures into a format the persistent system can deal with, tracking which objects were changed in order to propagate the updates onto disk, etc.), imposing higher development and maintenance costs.

Orthogonal Persistence (OP) [AM95, Atk78] is a language-independent model of persistence, defined by the following three principles.

❶ Type Orthogonality

❷ Persistence by Reachability

❸ Persistence Independence

These were first introduced by Atkinson *et al.* [Atk78, ACC82, ABC+83], summarised by Atkinson and Morrison [AM95], and their applicability to Java analysed by Atkinson, Jordan, and Printezis [JA98, AJ99b, Pri99a]. They are covered in more detail in sections 2.2.1, 2.2.2, and 2.2.3. Section 2.2.4 enumerates the benefits of OP over other persistence schemes and section 2.2.5 gives a brief historical overview.

2.2.1 Type Orthogonality

> *"Persistence is available for* all *data, irrespective of type."*

Many systems claim that they provide orthogonal (sometimes also referred to as *transparent*) persistence. However, deeper investigation usually reveals that rather than allowing "any data type" to persist, they allow "any data type, provided *X*", where *X* is: the data type has mapping/unmapping facilities defined on it, is a subclass of a persistent-object class, implements a certain persistence-related interface, etc. It can be argued that this is not OP, and definitely not transparent, since the application programmer has to decide for *each* data type whether it can persist or not and, in some cases, has to write the mapping code explicitly.

2.2.2 Persistence by Reachability

> *"The lifetime of all objects is determined by reachability from a designated set of root objects, the* Persistent Roots."

The concept of reachability is well understood by anybody who uses a programming language that relies on a garbage collector for its memory management [Jon96, Wil92]. It is only natural to extend this concept to also apply to persistence and provide a uniform model for short-lived and long-lived data. There are persistent systems that, even though they target a garbage-collected language, require explicit deletes in order to reclaim space in the storage system. This forces the programmer to use two different models of storage management and severely detracts from the benefit of introducing a garbage collector.

Persistence by reachability is also referred to as *Transitive Persistence.*

2.2.3 Persistence Independence

"It is indistinguishable whether code is operating on short-lived or long-lived data."

It is very often the case that, when programmers use similar data structures in memory and on disk, they have to replicate their code and keep one version "tied" to the required storage system. This has the disadvantages of the programmer having to maintain two versions of the code, the API of the storage system "getting in the way" of the algorithm that is being implemented, and, if a different storage system needs to be targetted, yet another version of the code needs to be created. Additionally, any third-party libraries have to be modified in order to operate over persistent data. In fact, this situation becomes worse since the transient and persistent versions of the code might not be able to co-exist in the same system (e.g. due to naming problems). The persistence independence principle allows for exactly the same code to operate over transient and persistent data. This eliminates the problems mentioned above and can minimise development time, while maximising code re-use.

2.2.4 Benefits of Orthogonal Persistence

Given the above three principles, the interaction between the programmer and the orthogonally persistent system is minimal. In fact, apart from the usual facilities that a programming language provides, the only extra persistence-specific calls needed are the following.

- **Register and retrieve the persistent roots** — It should be emphasised that persistent roots should *not* be associated with small object graphs, but rather with entire applications. Therefore, the calls that deal with persistent roots are invoked very rarely, typically only once inside the application-startup code. Code analysis by Grimstad *et al.* [GSAW98] supports this claim.
- **Perform a checkpoint** — This means that any changes performed on the data are atomically propagated to the disk and, in the event of a failure, they will be retrieved upon startup in a consistent state.

Another important benefit, which the architecture of orthogonally persistent systems usually allows, is incremental on-demand object-fetching. This allows them to be much more responsive, when they are initialised, and eliminates the *"Big Inhale"* problem [ABD+92], in which the initialisation of a system is slow since it has to first fetch all the data it needs.

It is interesting to note here that, from past experience, it is usually fairly difficult to explain to experienced programmers the concept of OP. This is because working with a traditional database involves a mapping between programming language data structures and the database model and the use of complex APIs through which the data is transferred. This is the model that seasoned programmers are used to and, when enquiring about OP, they are disappointed when they ask *"What does the API look like?"* or *"How do I run queries?"*, since the OP API is minimal and queries are not directly supported, as query engines should be built at the application-level, possibly using standard bulk-type libraries [CADA87]. In fact, it is reported by Liedtke that novice programmers absorb much more easily the concepts of OP [Lie93].

2.2.5 A Historical Note

Orthogonal persistence was first proposed by Atkinson in 1978 [Atk78] and its benefits are described in detail by Atkinson and Morrison [AM95].

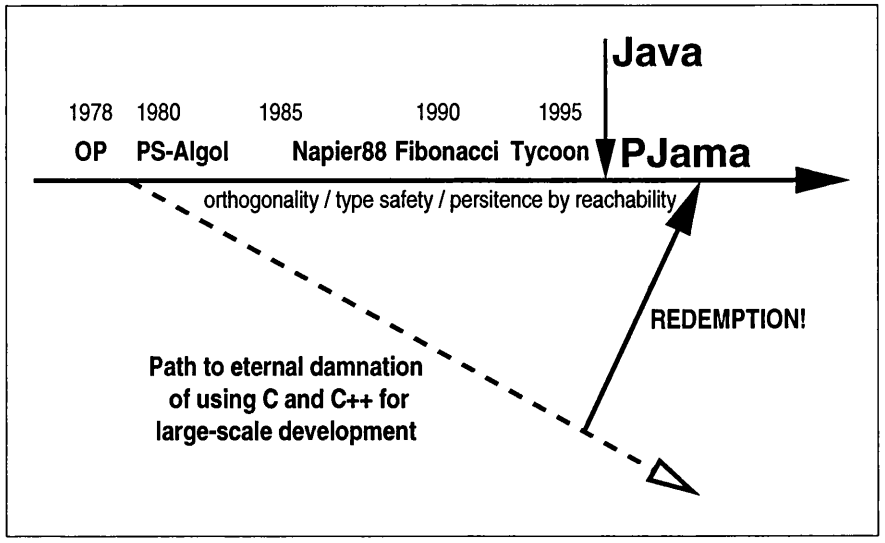


Figure 2.1: The "Java Event" and Orthogonally Persistent Systems

The first persistent programming language is considered to be Pascal/R [Sch77], an extension of Pascal that allowed relational queries inside the language. However, the first orthogonally persistent language was PS-Algol [ACC82], an extension of S-algol [Mor79], developed by the Universities of Edinburgh and St Andrews, Scotland, around 1980.

PS-Algol was succeeded by Napier88 [MCC+99], developed at the University of St Andrews, Scotland, that had a much richer type system and introduced parametric polymorphism and other implementation improvements. By this point, increased interest in orthogonally persistent systems yielded the languages Fibonacci [AGO99], developed at the University of Pisa, Italy and Tycoon [Mat99], at the University of Hamburg, Germany. In fact, the above three systems were developed as part of the FIDE (Fully Integrated Data Environments) and $FIDE_2$ ESPRIT projects. A book has been published on the contribution of these two projects [AW99].

It was only natural for the popularity of the Java language [GJS96, AG96] and the benefits of OP to be combined in the form of the PJama project, described in section 2.3. Figure 2.1 shows a timeline that follows the development of the systems mentioned above, current trends in software development, and how the "Java Event" affected them.

2.3 The PJama Project

PJama [ADJ+96, AJDS96, Jor96a, JA98, AJ99b, Pri99a] is a system that provides orthogonal persistence [AM95] for the Java programming language [GJS96]. It was developed collaboratively between the Department of Computing Science of the University of Glasgow and the Research Laboratories of Sun Microsystems Inc. It conforms to the three principles of OP, described in section 2.2, since

- instances of *any* Java class can persist (this includes the classes themselves, as they are instances of the class `Class`, and their static fields, GUI components, etc.),
- all objects reachable from a set of declared roots become persistent, and

- code which operates over transient data can also operate over persistent data, with no changes to the original sources (i.e. `.java` files) or post-processing of the compiled bytecodes (i.e. `.class` files) being necessary.

An additional requirement for the PJama system was to implement all the above without introducing any new constructs to or changing the syntax of the Java language. This way, third-party classes (even ones developed for "vanilla" Java) can persist unchanged using PJama [AJDS96, ADJ+96]. This was achieved by introducing a small number of PJama-specific classes that encompass the (minimal) API needed by the programmers to access the persistence facilities of PJama (⇝§2.3.2 and §2.3.3). This approach has also allowed the standard Java compiler (`javac`) to be used *unchanged* to compile PJama-targetted classes.

The PJama system achieves all the above by requiring a modified persistence-aware *Virtual Machine* (VM) (⇝§2.4 and §2.5). It can be argued that customising the VM is the only way to make some sets of classes persistent (e.g. `Class`, `Thread`, etc.), since their state cannot be accessed from Java and therefore a Java-only solution would be inappropriate. In fact, GemStone Inc. have taken the same approach for their GemStone/J product [Gem98b].

Unfortunately, the claim of complete type orthogonality is not yet entirely true. Even though the majority of classes can persist unchanged using PJama, a few of them either require some changes in order to persist or cannot persist at all. Examples of classes that require changes are ones which use the `transient` keyword in a manner incompatible with OP (⇝§2.3.4) or depend on static initialisers to load dymanic libraries. Examples of classes which cannot persist at all are ones tied closely to the implementation of the VM (`Thread`, `Exception`, etc.).

Issues of type orthogonality in Java are discussed in more detail by Jordan and Atkinson [Jor96a, JA98, AJ99b]. Currently, most of the obstacles that are in the way to complete orthogonality are technical issues and the PJama team remains committed to overcome them in the near future.

2.3.1 Making Classes Persistent

The PJama project has taken the approach of making the classes persistent along with the data. In this way, the application programmer does not have to keep track of which version of a class matches the persistent data in the store; an activity that can be complex and tedious for large projects. Instead, class-evolution tools can be used to evolve the classes in the store, safely and consistently. Even though this is a hard and complex problem, steady progress is being made as reported by Dmitriev [Dmi98, DA99b].

A class **C** becomes persistent if one of its instances becomes persistent or if it is used by another class **D** that has become persistent (e.g. if **D** accesses one of **C**'s methods or fields). When a class becomes persistent, its static fields also become persistent. If this was not the case, then any state in the static fields, which is shared by all instances, would be lost; this would force the class implementation to change, breaking the concept of persistence independence.

The only time that the static initialiser of a class is called is when the class is loaded from the file system. However, if it becomes persistent, the static initialiser is *not* called every time the class is fetched from the store, otherwise its static fields will potentially be re-initialised and their values might be inconsistent with the state of the persistent instances of that class. If some initialisation needs to be performed every time a class is fetched from the store (e.g. dynamic library loading), this can be done by the *Action Handlers* that PJama defines [Sun98f, JA98].

```
package org.opj.store;
import java.util.Enumeration;

interface PJStore {
   public void newPRoot(String, Object);
   public Object getPRoot(String);
   public void discardPRoot(String);
   public boolean existsPRoot(String);
   public Enumeration getAllPRootNames();

   public void stabilizeAll();
   ...
}
```

(a)

```
package org.opj.store;

class PJStoreImpl implements PJStore {
   static private PJStore store;
   static public PJStore getStore() {
      return store;
   }
   ...
}
```

(b)

Figure 2.2: The PJama API

2.3.2 The PJama API

The PJama API [Sun98f] is presented in figure 2.2.a. For simplicity and clarity, the exception-related lines have been omitted. All the necessary calls are declared in an interface called `PJStore`. The motivation behind this is to allow any third parties, who work on orthogonally persistent systems for Java, to use the same API and allow code to be portable across different systems. As explained in section 2.2.4, the API is minimal and only comprises a few calls to manage persistent roots, the `stabilizeAll` call, which performs the checkpoint operation mentioned in section 2.2.4, and a few other utilities (that are omitted from the figure).

The implementation class is called `PJStoreImpl` and is illustrated in figure 2.2.b. It implements the `PJStore` interface and it has a static field called `store` that represents the persistent store. This is initialised when the JVM is initialised and can be accessed by the programmer with the `getStore` static method. A concrete example of how the API is used is given in section 2.3.3. Currently, all of the PJama API is included in a package called `org.opj`.

It must be noted that the API summarised in this section corresponds to the one released in the PJama Classic system (⇝§2.4). It was altered slightly when ported to the PEVM system (⇝§2.5).

2.3.3 An Example of Using PJama

Figure 2.3 illustrates a concrete example of using PJama. Again, for simplicity and clarity, any exception-related lines have been omitted. Notice that all the PJama-specific code is included in grey boxes.

The class `ExtHashtable` (figure 2.3.a) is a subclass of the standard class `Hashtable` and introduces two new methods: `populate`, which populates the hash table in some manner (e.g. by reading a file or relying on user-input), and `display`, which displays the contents of the hash table.

The class `TransientCreateRead` (figure 2.3.b) illustrates how `ExtHashtable` is used. It creates an instance of `ExtHashtable`, populates it, and displays its contents. Of course, as in any transient program, the

```
import java.util.Hashtable;

class ExtHashtable extends Hashtable {
   public void populate() {
      /* populates the hash table */
      . . .
   }

   public void display() {
      /* display the contents of the hash table */
      . . .
   }
}
```

(a)

```
class TransientCreateRead {
   static public void main (String args[]) {
      ExtHashtable ht;

      ht = new ExtHashtable();
      ht.populate();
      ht.display();
   }
}
```

(b)

```
import org.opj.store.*;

class PersistentCreate {
   static public void main (String args[]) {
      PJStore PS = PJStoreImpl.getStore();
      ExtHashtable ht;

      ht = new ExtHashtable();
      PS.newPRoot("HashTable", ht);
      ht.populate();

      /* implicit stabilisation */
   }
}
```

(c)

```
import org.opj.store.*;

class PersistentRead {
   static public void main (String args[]) {
      PJStore PS = PJStoreImpl.getStore();
      ExtHashtable ht;

      ht = (ExtHashtable)
         PS.getPRoot("Hash Table");
      ht.display();
   }
}
```

(d)

Figure 2.3: A Concrete Example Using PJama

contents of the hash table cease to exist once the program finishes execution.

The class `PersistentCreate` (figure 2.3.c) illustrates how the contents of the hash table can be made persistent using PJama. After the hash table has been instantiated, it is registered as a persistent root, associated with the name "Hash Table", and then populated. Since PJama has an implicit checkpoint operation at the end of program execution, it will automatically propagate all persistent objects (in this case, the hash table and its contents) to the persistent store before exiting.

The class `PersistentRead` (figure 2.3.d) illustrates how persistent data can be retrieved using PJama. The persistent root, called "Hash Table", is looked up and this yields a reference to the `ExtHashtable` object. This can then be manipulated in the normal manner. It is worth pointing out that, at the point when the persistent root is retrieved, the entire hash table is *not* fetched into memory; only the parts of it that will be accessed are fetched, as and when they are needed.

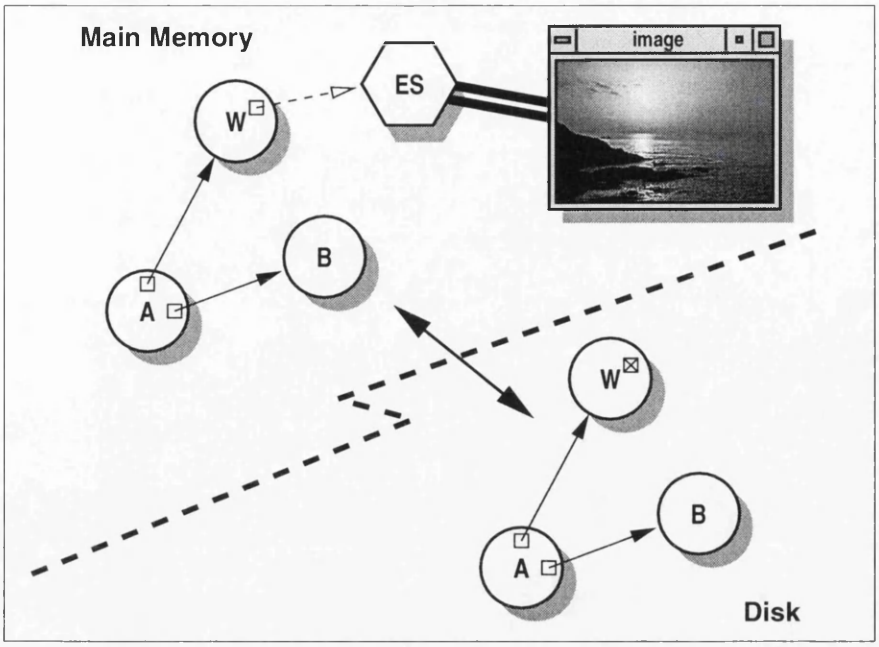


Figure 2.4: Handling External State in an Open Persistent System

In each example above notice that only three lines are PJama-specific (plus some exception handling that is not illustrated) and, in both cases, they are incorporated in a single class. This allows the `ExtHashtable` class to be used unchanged, even if it had been obtained from a third-party developer in bytecode-only form.

2.3.4 Open Persistent Systems

Early orthogonally persistent languages, like PS-algol [ACC82] and Napier88 [MCC+99], were designed as closed systems (the Tycoon system [Mat99] is a possible exception to this). This allowed their implementers to achieve complete type orthogonality, since the state of all the data that needed to persist was known to the system. This approach proved the feasibility of OP. However, it was very limiting, as the addition of any new facilities to the system, which had to communicate with external resources and therefore could not be written in the language itself, had to be incorporated inside the core of the system and could not be bundled as external libraries. This was the case, for example, for the windowing system and sockets. This limitation has been one of the biggest criticisms of orthogonally persistent systems.

The above has sparked an interest in orthogonally persistent systems that can handle external state through a well-defined interface. Such systems are called *open*. A severe complication arises however when an open persistent system attempts to make external state persistent. When it re-instates it, it will not know how to perform any required initialisation as it does not know the data contents, layout, and interpretation. The only way that this can be achieved in a generic manner is to delegate to the library that manages the external state the identification of which data shouldn't be put into the store and how its re-initialisation should be handled.

Data that the application or library chooses not to allow to become persistent will be referred to as *Transient*. An open persistent system should have a well-defined way of allowing programmers to mark data as transient and restore it to well-defined values, when moving it to/from disk, so that the next access to it can be detected and any necessary re-initialisation code run.

The handling of external state using transient data is illustrated in figure 2.4. Object **W** in memory contains the state of the image window in a format that the persistent system can understand and manipulate. Any changes to **W** are propagated, via calls to the appropriate library, to object **ES** that is the window state inside the windowing toolkit and is external to the persistent system. The reference from **W** to **ES** has been marked transient, hence the dashed arrow (notice that it is the reference field in object **W** that needs to be marked transient, *not* object **ES** itself). When object **W** is stored on disk, that reference will be cut and set to a default value. This will be detected, when **W** is re-activated, and the re-initialisation of **ES** will be triggered with a call to the windowing library.

From the beginning, the PJama system was designed to be open. The original intention was to use the `transient` keyword of Java [GJS96] to mark class fields as transient. However, the `transient` keyword is now widely used by Java developers in the context of *Java Object Serialisation* (JOS) [Sun98b] and its semantics are incompatible with PJama's needs. To deal with this, the PJama API was augmented with a new method called `markTransient` that allows programmers, and especially library developers, to "mark" class fields as transient for the purposes of the PJama system [Sun98f]. More information on the issues raised by introducing transient data in PJama, the incompatibility problems with JOS, and the inconsistencies in the current and widely-accepted definition of the `transient` keyword is given by Printezis *et al.* [PAJ99]. Additionally, Jordan and Atkinson provide further discussion on the issue of openness in persistent object systems [Jor96a, JA98, AJ99b].

2.3.5 A Historical Note

In 1994, the *Forest* project, led by Mick Jordan, was initiated at the *Research Laboratories of Sun Microsystems Inc.* (SunLabs). Its goal was to investigate advanced, scalable, and distributed configuration management techniques for software development environments, based on persistent object technology instead of the traditional, and apparently inappropriate, approach of using files, directories, or ad-hoc persistence schemes [JV95]. A prototype was developed, written in C++ and using ObjectStore [LLOW91] for the persistence facilities.

During 1995, it was obvious that Java was going to become popular, therefore the Forest team refocused their efforts on it and decided to reimplement their system in Java. However, no persistence mechanism was available at the time that did not have the pitfalls encountered when using ObjectStore (explicit free-space management, manual object clustering, etc.) [Jor96b]. Thus, in September 1995, the collaboration between SunLabs and the Department of Computing Science at the University of Glasgow was initiated with the goal to investigate the feasibility of introducing OP (↝§2.2) to the Java programming language. This project was named *PJava*, standing for *Persistent Java*. However, that name was later reserved by Sun Microsystems Inc. for their *Personal Java* product, therefore the project was later renamed to *PJama*. Meanwhile, the Forest team built a second prototype of their system, targetted for development using Java. It was written in Tcl/Tk [Ous93, Ous90], using files/directories for its persistence facilities, and it was a temporary measure until the PJama system was operational.

The first usable version of PJama was shipped to SunLabs from Glasgow, early in June 1996. It was based on Sun's JDK1.0 (its architecture is described in section 2.4) and it had very basic functionality. More features, like object cache replacement [DA97], explicit stabilisation during execution, a simple disk garbage collector [Pri96, Ham97], etc., were added to it and a port to JDK1.0.2 was done, before the first public PJama release, around January 1997. At the same time, the configuration management system developed

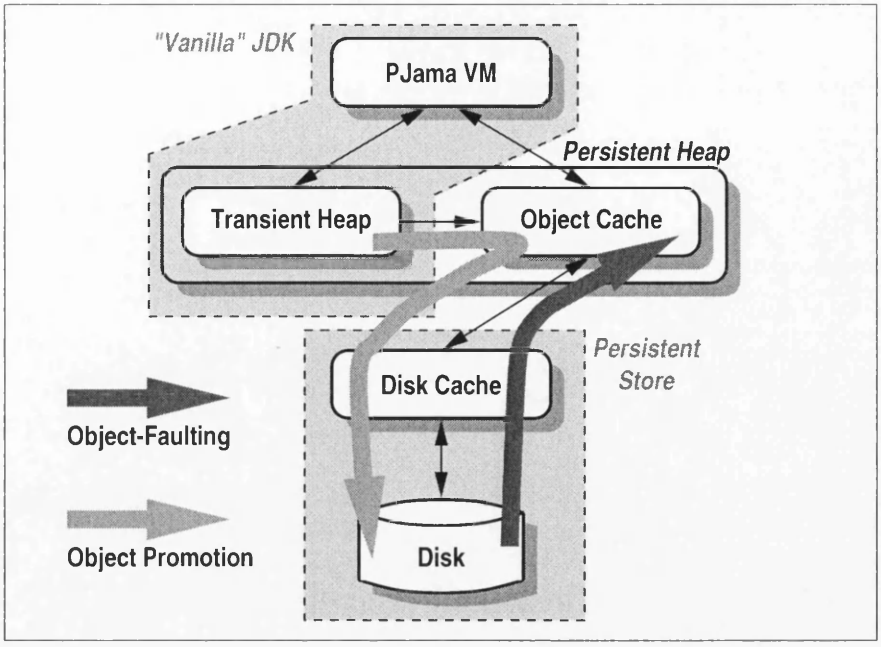


Figure 2.5: The PJama Classic Architecture

by the Forest group was being rewritten in Java, using PJama for its persistence facilities. It was eventually named *JP*, after a series of name changes. More information on JP and the configuration management work of the Forest project is given by Jordan, Van De Vanter, and Murer [JV95, JV97, Van98, VM99, MV99].

During 1997 and 1998, work on the core PJama system slowly migrated from Glasgow University to Sun-Labs. Ports to the 1.1.x versions of the JDK followed and the JDK-based PJama system started being referred to as *PJama Classic*. New facilities continued being added. These included distribution facilities [SA97, Spe98, Spe99] and schema evolution [Dmi98, DA99b]. The final JDK-based port of PJama was to JDK1.2. This proved more complicated than originally anticipated. Still, the corresponding public release shortly followed that of JDK1.2. By August 1999, 150 sites worldwide had downloaded the PJama Classic system.

Currently, PJama has been ported to EVM, a high-performance Java VM from Sun Microsystems Inc. (⇝§2.5). The resulting system, PEVM, is the first system to use Sphere (⇝§3 and §4), the new persistent object store from Glasgow University that has been under development since 1997. The use of Sphere in PEVM has eliminated most of the scalability problems present in PJama Classic (⇝§2.4.1) and has yielded a much more efficient and long-running system.

2.4 The PJama Classic Architecture

This section provides a brief description of the implementation of PJama Classic, which is currently based on Sun's JDK1.2. PJama Classic was initially built as a prototype to prove the feasibility of applying OP to the Java language. A high-level illustration of its architecture is given in figure 2.5.

The "vanilla" JDK platform comprises the VM and the *Transient Heap* (this is also known as the *Garbage*

Collected Heap), where objects are allocated and manipulated. The transient heap relies on a conservative garbage collector [BW88, Zor93] for its free-space management. Each object in the heap has a corresponding *Handle*, a small immutable structure that provides one level of indirection to the object. Any references to an object actually point to its handle, which in turn points to the object itself. This allows the object to be relocated during garbage collection, without having to keep track and "patch" reference fields in other objects or the runtime stacks that point to it. This is essential since such fields will have been discovered "conservatively" and in fact may not be object references [BW88].

As illustrated in figure 2.5, PJama Classic extends the basic JDK architecture by adding the *Object Cache*, where persistent objects are cached and manipulated, and the *Persistent Store*, used for long-term and fault-tolerant object storage. Objects in the transient heap and in the object cache appear the same to the VM; therefore the combination of the transient heap and the object cache can be viewed as a single *Persistent Heap*. In fact, these two components were unified in PEVM, a subsequent implementation of PJama (⇝§2.5). When the object cache fills up, some objects in it are "evicted", making space for others to be brought-in [DA97]. PJama Classic has adopted a *no-steal policy* on the object cache, i.e. dirty objects are not allowed to be automatically evicted, until a stabilisation operation (see below) has propagated the changes on them to disk and has reverted their state to clean.

The persistent store used in PJama Classic is a simple custom-built physically-addressed one. The PID of every object (i.e. the "address" of every object on disk, ⇝§3.4.1) is the physical offset of the object in the store. Pages of the store are faulted-in, as necessary, and are cached on a fixed-size *Disk Cache*, running the *Clock* algorithm for page-eviction [Tan92]. The store relies on the *Recoverable Virtual Memory* (RVM) system for its logging facilities to ensure fault-tolerance [MS94]. RVM has imposed a no-steal policy to the disk cache also; i.e. no pages with uncommitted changes on them are allowed to be automatically evicted, until a stabilisation operation is initiated (see below).

When an object needs to be fetched from disk, the page where it resides is copied to the disk cache and then the object is copied to the object cache (this operation is called *Object-Faulting* [Bro89, HMB90, WD92, HM93a, HM93b] and is illustrated in figure 2.5). At this point, all the references in it are PIDs. The VM will detect when one of these is dereferenced, fetch the appropriate object from disk, and replace the PID with the memory address of the newly fetched object. This operation is called *Pointer Swizzling* [Bro89, Mos90b, WD92, WK92, HM93a, DA97]. The detection takes place by small pieces of code, usually referred to as *Residency Checks* [Bro89, HMB90, HM93a, HM93b, LM99], planted in appropriate places throughout the VM. To avoid fetching the same object twice, an in-memory table, called the *Resident Object Table* (ROT) [Bro89, Mos90a, DA97], keeps track of which objects are resident and where in the object cache they reside (it essentially maps PIDs to memory addresses).

Stabilisation is the operation that propagates updates on persistent objects from the object cache to the store. It is initiated either by the program (*Explicit Stabilisation*) or automatically at the end of the program execution (*Implicit Stabilisation*), provided the program is exiting gracefully. The VM automatically keeps track of which objects are updated (dirtied) during execution [HBM93, LM99]. During stabilisation, the updates on all dirty objects are propagated to the store. This happens in terms of a single RVM transaction, which ensures that the updates are atomic. PJama Classic only supports a global stabilisation operation. Work is underway to investigate the feasibility of partial user/programmer-defined stabilisations (essentially introducing transactions inside the VM) [Day96, DAV97]. This facility however is unlikely to ever be fully implemented in PJama Classic.

When, during stabilisation, objects in the transient heap become persistent (because there are references to them from persistent objects in the object cache), they are moved to the object cache and an image of each of them is written to disk, via the disk cache. This operation is called *Object Promotion* (↝§5) and is illustrated in figure 2.5.

The above architecture and the tight coupling between the components allows the performance of the PJama Classic system to be very good, since all the object-fetching and dirty object-tracking is done entirely inside the runtime system. In fact, third-party performance evaluation experiments by Ridgway *et al.* show PJama faster than all the approaches written entirely in the Java language that map objects to relational or object-oriented databases [RTW97].

More information on the memory management of PJama Classic is given by Daynès and Atkinson [DA97].

2.4.1 Scalability Issues

Extensive use of PJama Classic for almost three years has revealed some scalability problems. These are enumerated below.

❶ **Persistent Object Updates** — The number of persistent objects that can be updated per stabilisation is limited due to the imposed no-steal policy on the disk and object caches (↝§2.4). This policy requires all updated objects to be resident in the object cache and their corresponding store pages to be resident in the disk cache until the stabilisation operation has been committed. This effectively imposes at least equal size-requirements on both caches that are linearly proportional to the volume of updated objects. Because data is never properly clustered, a larger disk cache is typically required.

❷ **Physically-Addressed Store** — As described in section 2.4, the PJama Classic store is physically-addressed. This renders on-line object relocation (and hence any concurrent relocating disk garbage collection or reclustering schemes) virtually impossible, since if an object is relocated, its PID will change. On-line PID changes are inherently difficult to handle as all appearances of each updated PID in all the objects in the store have to be updated, as well as in objects in the object cache and anywhere else in the system where they might be stored (e.g. the ROT). Any approach that allows PIDs to change complicates the overall system architecture considerably.

❸ **Memory Management** — The conservative memory management of PJama Classic (↝§2.4) has proved troublesome to work with. The absence of exact information on where pointers are stored on the runtime stacks has complicated the stabilisation operation and the cache eviction mechanism, both of which need to move objects that are potentially directly referenced from the stacks[4] [DA97]. For the above reason, it is doubtful whether any advanced high-performance memory management techniques can be implemented in such a conservative environment.

❹ **Object Promotion** — The object promotion mechanism of PJama Classic (↝§5.4.3) imposes equal size-requirements on the transient heap and object cache that are linearly proportional to the volume of promoted objects. Because promoted objects are copied from the transient heap to the object cache, enough space in both of them must exist to store all promoted objects, essentially doubling the promotion memory-requirements and introducing copying costs.

[4]Even though all object references in PJama Classic point to the object's handle, as explained in section 2.4, direct references to objects can still be cached on the runtime C stacks.

Items ❶ and ❷ reflect limitations of the persistent store, item ❸ reflects limitations of the memory management framework, inherited from the original JDK architecture, and item ❹ reflects a combination of the above.

2.5 The PEVM Implementation

To overcome the deficiencies of PJama Classic, presented in section 2.4.1, a new VM was sought to be used as a basis for a high-performance PJama implementation. *EVM* from Sun Microsystems Inc. was chosen for this purpose. The storage technology chosen to be coupled with EVM was Sphere, the POS described in chapters 3 and 4. Sphere was specifically designed to overcome the store-imposed limitations of PJama Classic, presented in section 2.4.1. The PJama implementation based on EVM and Sphere will be referred to as *PEVM*.

The name EVM is an abbreviation of *ExactVM*, referring to its ability to support *exact* memory management [AD97], i.e. all the pointers on the runtime stacks and objects are known to the garbage collector (such an approach is also called *non-conservative* or *accurate* [Jon96]). EVM has been publically released by Sun, referred to as *JDK1.2 for Solaris, Production Release*[5]. Its main strengths are presented below.

- **Advanced Exact Memory Management** — The memory management component of EVM is abstracted from the rest of the system and is accessed through a well-defined interface, called the *GC Interface* [WG99, ADM98]. This allows for its implementation to be easily changed and different garbage collection algorithms to be implemented and evaluated. For the purposes of PEVM, this is a very useful feature since it allows for different policies of cache and heap management to be implemented and evaluated relatively easily. Additionally, the knowledge of the location of references on stacks [AD97] and the implementation of GC safe-points [Age98] contribute to the simplification of the stabilisation and cache eviction operations.

- **High-Performance Object Synchronisation** — EVM has adopted an advanced synchronisation mechanism for the purposes of MT-safety. Its main advantages are its efficiency and scalability when a large number of threads and CPUs are used. This mechanism is described by Agesen *et al.* [ADG+99a, ADG+99b]. This allows server-like applications running on PEVM that need to serve large numbers of connections to run efficiently and have high throughput.

- **High-Performance JIT Compiler** — A JIT compiler is included in EVM that improves its performance considerably, compared to the purely-interpreted version of the system [DA99a]. In PEVM, the JIT has been modified to include the necessary residency checks, described in section 2.4, inlined within the generated native code. This has the potential to allow technology developed by Hosking *et al.* [HNCB98, BNHC98], which uses heuristics to eliminate residency checks, to be included in the JIT and avoid the extra step of bytecode post-processing originally proposed [BNHC98].

2.6 Future Work

Currently, the PJama project is focusing its efforts on the PEVM implementation, of which the first version is already operational. Possible research areas that will be explored, using PEVM as the implementation

[5]Interestingly enough, EVM has undergone yet another name change. In Autumn 1999 it was renamed *SunLabs Virtual Machine for Research* (ResearchVM) and was retargetted to purely research purposes. The well-known and hyped *HotSpot*™ technology succeeded it as Sun's high-performance production Java VM. However, the name EVM will be used for the rest of this thesis.

platform, are

- concurrent cyclic disk garbage collection (↝§6);
- concurrent archiving facilities;
- type-safe schema evolution [Dmi98, DA99b];
- advanced memory management and incremental object-cache replacement;
- elimination of residency checks [Hos96, CH97, HNCB98, BNHC98, Bra98, Nys98, NHW+99]; and
- persistence of computation (threads) [JA99].

Based on experience gained implementing and using the PJama system, the PJama project is also exploring the theoretical issues that arise from the introduction of OP to the Java language. The *Orthogonal Persistence for the Java Platform Proposal*, edited by Jordan and Atkinson [JA99], takes the first official step needed to introduce OP to the Java language as standard. It includes the introduction of all necessary API calls in the standard Java classes, complete type orthogonality, persistence of threads, and other related issues. A prototype implementation of this system is under way, based on the JavaInJava VM, a VM written entirely in the Java language [Tai98], and using PJama for its persistence facilities.

Additionally, Jordan and Atkinson have identified issues concerning complete type orthogonality in PJama [Jor96a, JA98, AJ99b] and Printezis *et al.* have identified inconsistencies in the definition of the `transient` keyword in Java and have proposed ways to eliminate them [PAJ99]. These areas need to be explored further.

Finally, another direction that the PJama project would eventually like to follow is the introduction of multiple, extensible, and customisable transactions inside the same PJama VM [Day96, DAV96, DAV97]. The introduction of such transactions to persistent systems has always been considered a difficult task [BZ98, BDN+98]. Still, steady progress is currently being made, lead by Daynès, and several prototypes have been built, based on the JavaInJava VM, PJama Classic, and currently PEVM.

2.7 Other Persistence Solutions for Java

The feasibility of Java objects persisting is discussed by Morrison *et al.* [MCKM96] and different approaches to achieve this are overviewed by Moss and Hosking [MH96].

Since the 1.1 release of Sun's JDK, *Java™ Object Serialisation* or *JOS* [Sun98b] has been considered to be the default persistence mechanism for the Java language. JOS is now part of the standard `java.io` package and provides facilities to serialise/deserialise object graphs to and from bytestreams. The generated bytestreams can either be written to disk or sent to another machine over the network (the latter being the original use for JOS). There is a number of problems with JOS being used to provide production-quality persistence for Java: it has no transactional support, individual objects on the bytestream cannot be updated without the entire bytestream being recreated, it suffers from the "Big Inhale" problem described above (as the bytestream must be entirely deserialised before any objects from it can be used), etc. However, the biggest criticism is that JOS is not orthogonal since, for an instance to be allowed to be serialised, its class must implement the `java.io.Serializable` interface. Many of the core classes do not do this. This breaks

the principles of orthogonality and persistence by reachability. The disadvantages of JOS are discussed further by Evans [Eva00] and D. Jordan [Jor99a].

JavaSpaces™ [Sun98a] is a product from Sun Microsystems Inc. that implements persistent tuple-spaces. It was created to provide persistence facilities to the Jini™ platform [AWO+99, Wal99]. Its current implementation is based on JOS.

The first persistence-related API for Java developed by Sun Microsystems Inc. was JDBC™ [Sun98e], an API that allows access to relational databases via SQL queries [MS93]. Obviously this approach, even though convenient when accessing legacy data, is far from transparent or even portable, since it is well-known that different relational databases implement their own version of SQL. Still, all the major database vendors support JDBC and provide the necessary drivers for their particular architecture.

Another approach to access relational databases from Java via SQL is provided by the *SQLJ* standard [CSH+98]. This is a language extension to Java that allows the programmer to write static SQL queries entirely embedded in Java and also to execute Java code from inside SQL queries. This is achieved by a preprocessor that processes the SQLJ source and generates pure Java source. The database access in the generated code is accomplished via JDBC. Even though SQLJ is easier and more intuitive to use than JDCB, it also inherits the incompatibilities between different databases that JDBC has.

An aggressive approach for Java support has been taken by Oracle Corp. Version 8 of its system contains *JServer* [Orawww], an integrated custom-built, high-performance, and scalable Java VM that provides Java services (a CORBA 2.0 ORB, an embedded server-side JDBC driver, an SQLJ translator, etc.). Oracle has also embraced the *object-relational* model [SBM96] that maps Java objects to relational tables.

Another category of systems that provide persistence for Java includes *ObjectStore/PSE* [LLB+97, Obj99] (written entirely in Java) and the systems that conform to the *ODMG standard* [Cat97] for object-oriented databases, such as products from Versant Corp. [Verwww], POET Software [POEwww], Objectivity Inc. [Objwwwa], etc. These also claim to implement orthogonal persistence by reachability. However, the way this is achieved is by post-processing the Java class files to include "transparent" extra calls to the persistence-related classes. This results in requiring two versions of the classes (one for transient and one for persistent use), which complicates their management and introduces extra complexity for the programmer. For example, the popular JGL collection classes [Objwwwb] are available in two versions, standard and "annotated" for use with ObjectStore/PSE. If this trend spreads further, it will become intolerable. Furthermore, these systems actually do impose limitations on which classes they will allow to persist. An example of this is ObjectStore/PSE that provides its own hash table implementation, since it has problems dealing with the standard `java.util.Hashtable` class. Other orthogonality problems of the ODMG standard are discussed further by Alagić [Ala97].

SuperCede™ — *Professional Edition* is a Java development environment from SuperCede Inc.[6] [Sup98b]. Apart from the standard facilities that similar products usually provide (grouping classes into projects, interactive GUI builder, facility to generate native executables, etc.), SuperCede also includes a very powerful debugger that allows the programmer to dynamically and safely modify classes during execution, usually without the need to stop and restart the application (this depends on the changes themselves and how they affect the state of the runtime stack). Additionally, SuperCede is distributed with the *PersistentMemory*™ technology that provides the programmer with a transactional persistent heap [Sup98a], very similar in

[6]Recently, the Java technology developed by SuperCede Inc. was acquired by Instantiations Inc.

features and operation to the Texas persistent store [SKW92, Kak98, WK92]. An object is designated as persistent at allocation-time and the programmer specifies this by using one of the extra new operators that SuperCede extends the standard Java language with. Access to the transactional facilities is provided via the PersistentMemory API [Sup98a].

The *GemStone/J* product from GemStone Inc. [Oti96, Gem98b, Gem98a] has taken a similar approach to the PJama system by implementing persistence by reachability for Java and requiring a custom-modified Java VM. It employs a client-server model where several VMs can connect to the same persistent server where the objects are stored. Objects can be shared by different VMs and access restrictions are provided. GemStone/J is also marketed as a middle-tier solution, i.e. to provide persistent caching facilities for (much slower) RDBs. Unfortunately, GemStone/J is not orthogonal because static fields are not made persistent as the semantics of this would have been difficult to define in a multi-user environment.

Finally, some very specialised schemes that provide persistence for Java have also been reported, such as virtual memory *checkpointing* [How98] or using *persistent address spaces* [DHF96] in the context of a persistent operating system [DdF+94].

2.8 Summary

This chapter gave an overview of the PJama system, an orthogonally persistent environment for the Java language. It covered the Java language and its history (⇝§2.1), orthogonal persistence and its history (⇝§2.2), the goals of the PJama system (⇝§2.3) and its architecture (⇝§2.4 and §2.5), and its potential future directions (⇝§2.6). It then enumerated other persistence solutions for Java (⇝§2.7).

The next chapter (chapter 3) presents the design of Sphere, the persistent object store designed to overcome the store-imposed scalability problems of PJama Classic that were described in section 2.4.1.

Everything that has transpired has done so according to my design.
*— **Emperor Palpatine**, Ruler of the Empire*

Chapter 3

The Design of Sphere

This chapter describes the design of Sphere, a persistent object store developed at the Department of Computing Science of the University of Glasgow, Scotland. The name Sphere stands for **S**torage for **P**ersistent **H**ierarchical **E**nvironments with **R**ecoverable **E**xecution. It also refers to the adaptability of Sphere, as a sphere will always land in the same way, after it is thrown even in the most hazardous terrain. Sphere was originally called *PJSL*, the **PJ**ama **S**tore **L**ayer.

Section 3.1 defines the mutator, the application layer or layers that create load for the store, and section 3.2 enumerates the requirements imposed on the design and implementation of Sphere. Section 3.3 describes the differences between object-based and page-based stores and explains why the former approach was adopted. Section 3.4 presents the three-level nesting of containers in Sphere. The three main abstractions, object kinds, partition regimes and descriptors, are described in section 3.5. Section 3.6 presents how partitions in Sphere are managed, while section 3.7 outlines the Sphere logical object addressing scheme. Section 3.8 presents the two-level free-space management scheme adopted in Sphere. Section 3.9 concludes the chapter.

A slightly different version of this chapter has been published in the Proceedings of the Second International Workshop on Persistence and Java, Half Moon Bay, CA, USA, August 1997 [PAD+97a], as well as a technical report [PAD+97b].

3.1 The Mutator

For the remainder of this thesis, the application that creates the load for Sphere will be referred to as the *Mutator*. Figure 3.1 illustrates three possible examples of mutators. These are described below.

- **Example A** — The mutator is an application written in C that accesses Sphere directly through its public API (⇝§A). Examples of such applications are the test harnesses used to obtain the measurements presented throughout this thesis (⇝§5.3 and §6.4).
- **Example B** — The mutator is an application written in a language other than C. It accesses Sphere through an adaptor that provides the language with an interface to C and hence allows it to communicate with the Sphere public API. An example of such a mutator was an experimental pickling system

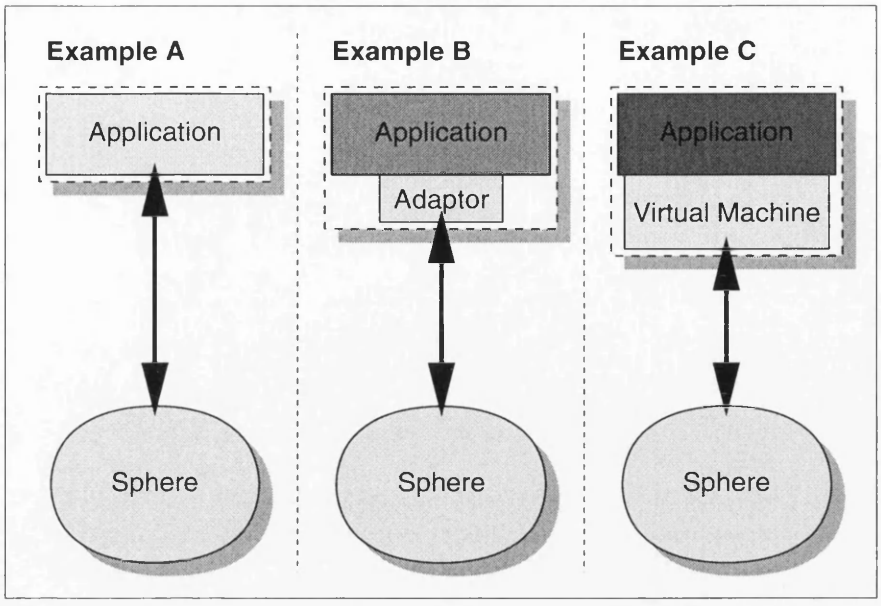


Figure 3.1: Examples of Mutators

[BNOW93] for the functional programming language Haskell [PHA+96] that was implemented at the University of Glasgow and relied on an early version of Sphere for its persistence facilities.

- **Example C** — The mutator is an application written in an interpreted language (e.g. Java [GJS96]). It accesses Sphere through the VM that interprets it. Examples of such applications are Java programs executed by the PJama VM (⇝§2).

The above three examples are not by any means the only situations in which Sphere can be used and more complicated usages can be envisaged; e.g. a client-server application where multiple clients communicate with a single Sphere-based server. However, notice that the mutator is not necessarily the only layer that directly communicates with the Sphere public API but instead the entire system that operates "on top" of Sphere. For example, in the case of example **C**, the mutator is considered to be the combination of the VM and the application it executes; this is because both contribute towards creating the load for Sphere: the VM typically defines how objects will be mapped to Sphere and the application generates the access patterns.

3.2 Requirements

The primary requirement of Sphere is to support the operation of a high-performance PJama implementation (currently PEVM, ⇝§2.5) when running real workloads. The typical workload makes long-running and complex use of highly-structured data, such as that connected with software construction [JV97]. Ultimately, future version of the PJama system will require concurrent access to the store to be organised as long running and flexible transactions [Day96]. It is hoped that the design of Sphere (presented in this chapter) and its implementation (⇝§4) are sufficiently general to service a wide range of applications and support various language implementations and stand-alone applications. Its flexibility should also allow for a series of store implementation experiments.

The specific and immediate design goals for Sphere were the following.

- Support for complete orthogonality, so that *any* object type can be accommodated, irrespective of their structure and size.
- The ability to reason about an object's layout without relying on up-calls to the mutator. For example, the ability to find all reference fields inside each stored object in order to work fully off-line, if necessary.
- The accommodation of at least 10GB of highly-structured data, typically dominated by large numbers of small objects.
- Minimal dependence on architecture- and operating-system-specific facilities to ease portability and allow Sphere to easily operate directly on hardware without the presence of an operating system[1].
- The capability of operating over file systems or raw disks and taking advantage of multiple hard drives.
- The introduction of a new recovery mechanism that does not have the limitations imposed by RVM [MS94] on the PJama Classic system (⇝§2.4.1).
- A well-defined interface between Sphere and the mutator, in order to increase Sphere's maintainability and adaptability.
- The ability to operate safely in a multi-threaded environment. This requirement applies to most Sphere operations, apart from concurrent updates to the same object; these are assumed to be serialised by the mutator (⇝§4.12).
- Facilities to support schema evolution on the stored objects.
- Support for a concurrent transactional load from the mutator.
- The inclusion of an incremental garbage collector which automatically discards all objects that are not transitively reachable from a given persistent root.
- The capability of continuous operation with incremental algorithms for its management operations (garbage collection, archiving, etc.).

These requirements were drawn up from the PJama team's experiences of using the PJama Classic system and other persistent systems (e.g. Napier88 [MCC+99], PS-algol [ACC82], and the combination of C++ and ObjectStore [LLOW91]) and from discussions with other users and researchers. The remainder of this chapter concentrates on the fundamental design decisions that were taken to meet the above requirements.

3.3 Object-Based vs. Page-Based Stores

Persistent object stores fall into the following two categories.

1. **Page-Based** — In these stores, objects are grouped in physical units, referred to as *pages*, and are manipulated (fetched, updated. etc.) together directly on the page and not copied to a dedicated object cache. The addressing model they adopt is typically a physical one, i.e. the address of each object reflects its location inside the store. Examples of such stores are ObjectStore [LLOW91], Texas [SKW92], etc.

[1] This was required as there was a plan to create a version of PJama that ran on the "bare metal".

❷ **Object-Based** — In these stores, objects can be manipulated individually (⇝§3.4.1), usually after having been copied to an object cache, and the addressing model adopted is flexible: either physical or logical. Examples of such stores are the Napier store [BR91], the storage system of GemStone/J [Gem98b], etc.

The main advantage of page-based stores is that the page size where objects are grouped also corresponds to the unit of transfer between the operating system and the disk. In this way, if several objects on a single page are used roughly at the same time and are needed roughly for the same duration, page-based stores can operate very efficiently, as only a single interaction with the store is necessary in order to manipulate several objects. On the other hand, if the object clustering is less than perfect, a few popular objects can cause the corresponding pages to be kept in memory, while the rest of the objects on them are not used, leaving less memory for further object fetches and eventually comprimising performance[2].

Object-based stores ameliorate the above problem by copying the popular objects to the object cache and allowing the corresponding pages to be evicted as necessary. Additionally, they provide extra flexibility by allowing a logical addressing model. This allows objects to be moved without having to patch potentially a very large number of reference fields, as their identity does not change. This can facilitate several operations:

- the store garbage collector can move objects for compaction purposes (⇝§6.2),
- the schema evolution mechanism can efficiently change the contents, layout, and size of an object (⇝§6.7), and
- any reclustering facilities can dynamically move objects to improve object clustering and hence optimise performance.

For the above reasons, it was decided to adopt an object-based approach when designing Sphere.

3.4 Nesting of Containers

This section introduces the hierarchy of container entities included in Sphere. Section 3.4.1 introduces objects, the unit of transfer between Sphere and the mutator, section 3.4.2 introduces pages, the unit of transfer between Sphere and the disk, section 3.4.3 introduces partitions, which are object containers, and section 3.4.4 introduces segments, which are partition containers.

3.4.1 Objects

An *Object* in Sphere is defined as a container for one or more *fields*. A field can be a 1, 2, 3, 4, or 8-byte scalar or a currently 4-byte reference[3]. This definition not only includes full-fledged objects, with encapsulation and inheritance in the context of object-oriented programming, such as C++ and Java `class` types, but is generic enough to also include any kind of structured data records, such as Pascal `record` types and C `struct` types [Wil92].

[2]This was a notorious problem, that developers could easily come across when using the ObjectStore system. This problem could be aggrevated by careless implementation of the manual object clustering that ObjectStore requires [Jor96b]

[3]The PEVM system is currently using 4-byte references. However, Sphere can be recompiled to support 8-byte references, if necessary.

The structure of an object should be unambiguously defined and known to Sphere[4]. The size of an object can vary up to a current maximum of 2^{27} bytes (section 4.5.4 describes the reasons for this limitation) and is set at object-allocation time. Only the schema evolution facilities (⇝§6.7) can alter the structure and/or size of an object, otherwise these remain fixed throughout its lifetime.

Each object has a unique identity, called its *Persistent Identifier* (PID). The PID can be seen as the "address" of the object in the store, even though in Sphere the addressing model is logical rather than physical (⇝§3.7.1). Given its PID, the mutator can manipulate an object via a set of operations: allocation, (whole or partial) fetches, and (whole, partial, or field) updates[5]. These are described in detail in appendix A. Notice that there is no explicit de-allocation operation, as Sphere relies on a garbage collector for automatic space-reclamation (⇝§6). PIDs of objects that have been de-allocated may be recycled.

Each Sphere store has a single object that serves as the persistent root of that store (⇝§2.2.2). The PID of this object is retrieved via the `sphere_getPersistentRoot` API call (⇝§A.2.6). All other PIDs are retrieved from reference fields inside other objects, when these are fetched from the store. Some POSs actually maintain multiple named persistent roots. Sphere only maintains one and it is up to the mutator to implement, if necessary, the management of multiple roots (PEVM does this).

3.4.2 Pages

As described in the previous section, objects are the unit of transfer between Sphere and the mutator. However, due to their highly varied size and their expected small average size, they are not appropriate to be the unit of transfer between Sphere and the I/O devices. A more appropriate unit for this would be the unit of transfer the operating system uses to access these devices. This is typically called a *Page*.

The size of such a page is usually 4K (SPARCstation 20, Solaris 2.5.1) or 8K (Sun Ultra 2 Creator, Solaris 7). In Sphere the latter is used. However, it has been shown that, in the context of persistent object stores, such page sizes are fairly small and greater performance can be achieved by using larger units of transfer (⇝§5.3.5). For this reason, pages in Sphere are assumed to be larger than the operating system page size and experiments will eventually be necessary to decide on an optimal size[6]. Currently the Sphere page size is set to 8K.

Due to their varied size, objects might or might not fit in their entirety on a single page. Objects that do are referred to as *Small Objects* and those that do not are referred to as *Large Objects*. Typically, in POSs, the vast majority of objects are small, as early experiments with PJama_0 [Pri96, Ham97] as well as measuring Napier88 [PC96] have shown.

[4]Currently, only the location of reference fields that contain PIDs need to be known so that Sphere can perform the marking phase of disk garbage collection (⇝§6) and be able to identify reference-field updates in order to manage the reference counts of objects (⇝§4.10). In the future it might be necessary for Sphere to access information on the size and type of each field, so that it can perform translations when running on a machine of different endianness.

[5]Even though objects are considered to be the unit of transfer between Sphere and the mutator, the partial operations are provided for efficiency reasons.

[6]Originally, it was planned to call them *Transfer Units* in order to distinguish them from operating system pages. However, it was eventually decided to call them pages anyway, as it is a term easily understood by researchers on the field.

3.4.3 Partitions

One of the fundamental decisions taken early into the Sphere design phase was to group objects into *Partitions* [YNY94]. Even though this complicates the management of the store, it has several important advantages over using a flat-space store, where all objects are managed by global free-space management.

- Partitions can be treated as garbage collection units so that the garbage collector does not have to visit the entire store, but only parts of it, one at a time [YNY94, AFG95].
- Grouping together objects of different size/behaviour provides the opportunity to apply different management techniques to different partitions, according to their contents (↝§3.5.1 and §3.5.4).
- It also turns out that partitions contribute to the efficiency of the schema evolution facilities of Sphere. The evolution mechanism can identify partitions that contain evolving objects and ignore all others, thus avoiding excessive and unnecessary scanning (↝§6.7). It then evolves one partition at a time, limiting space requirements [HAD99].

Partitions vary in size from a few megabytes up to the current maximum of one gigabyte (↝§4.8.1), depending on their contents.

Given all this, partitions can be seen as smart object containers, that are optimised for the kind of objects they contain. However, it must be pointed out that, in order to simplify the use of Sphere, the presence of partitions is hidden from the mutator and is internal to Sphere (↝§3.7.2).

3.4.4 Segments

As mentioned in section 3.2, a single Sphere store may be required to span multiple hard drives and to operate over a mixture of files and raw disks. To localise the impact of this requirement, a new abstraction was introduced. A Sphere store comprises one or more *Segments*, each of them containing a number of partitions. Each segment can reside either on a raw disk or on a file in the file system[7]. The partition allocation and addressing operations (↝§4.6.3 and §4.6.2 respectively) abstract away from the presence, number, and size of segments, hence only a limited portion of the Sphere code needs to operate over segments explicitly.

Segments were originally introduced for pragmatic reasons. The `lseek` operation of Solaris 2.5.1 accepted a signed 32-bit integer as the offset of the file/raw disk that was being addressed, imposing a 2GB limit on the size of such entities. The introduction of multiple segments cured this problem, as only each segment had such a size limit imposed on it. More recent versions of Solaris (i.e. Solaris 7) provide alternative `lseek` calls that accept a 64-bit integer as the offset (`lseek64`), essentially eliminating the above problem. However, the segment abstraction was retained in Sphere because it provides a straightforward and portable way to

- use multiple hard drives (by placing segments on different drives rather than relying on operating system facilities, e.g. virtual raw partitions), and
- to extend the store (by adding new segments).

The current maximum number of segments in a single Sphere store has been arbitrarily set to 32 (↝§4.6.1). Segment sizes are typically much larger than the partitions they contain, up to a current maximum of 2GB because of the reasons mentioned above.

[7] Access to hard drives on remote machines is currently only supported via NFS [Sun89].

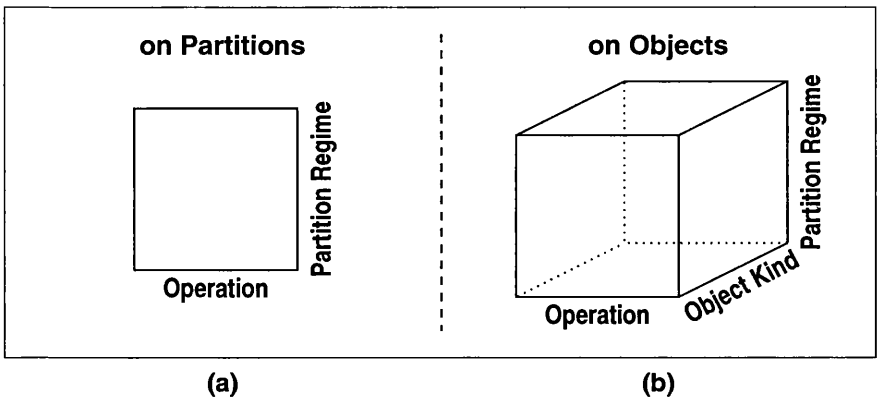


Figure 3.2: Operation Spaces

3.5 Abstractions

This section presents the main abstractions introduced in Sphere. Section 3.5.1 introduces partition regimes, which abstract over the management of partitions, section 3.5.2 introduces object kinds, which abstract over the management of objects, and section 3.5.3 introduces descriptors, special objects that are used to describe the structure of objects. Finally, sections 3.5.4 and 3.5.5 discuss the impact that these abstractions have on object clustering and the dataflow between Sphere and the mutator, respectively.

3.5.1 Partition Regimes

Managing free-space within a partition can be achieved in many ways: compaction, free-lists, bitmaps, etc. (⇝§3.8.1). The combination of the free-space management scheme, along with some additional organisation parameters, will be referred to as a *Partition Regime*. Partitions of the same regime will have the same internal structure and will usually contain similar (in structure, size, behaviour, etc.) objects. One regime can be more appropriate than another for certain kinds of objects, therefore several regimes can co-exist in the same store, applied to different partitions. Some examples of why different partitions might need to be managed in different ways are presented below.

- Introducing regimes specifically for small objects allows the elimination of any boundary checks, as small objects are guaranteed to fit entirely within a single page. Optimising the management of small objects is important since they are expected to be the majority of objects in a POS. Experiments with PJama Classic [Pri96, Ham97] and Napier88 [PC96] confirm this.

- Even though using compaction for free-space management is a proven way to eliminate fragmentation [Wil92, Jon96], it is also known to impose excessive copying in cases where large objects are likely to survive many garbage collections. For this reason, some compacting garbage collectors often provide a *large-object area* that is managed with in-place de-allocation [Jon96]. In a similar manner, large-object regimes can be organised to provide in-place de-allocation, while small-object regimes can benefit from compaction (⇝§3.8.3).

- Partitions containing only large scalar objects (commonly images, sound samples, and numeric data) can be allowed to grow very large without compromising garbage collection times. This is because such objects do not need to be scanned; instead the information on cross-partition references into the partition (i.e. reference counts, ⇝§4.10) can solely determine whether or not they are reachable.

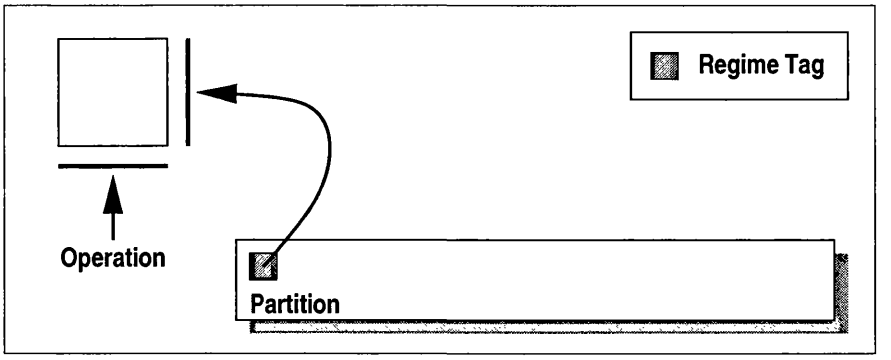


Figure 3.3: Invoking an Operation on a Partition

Based on the above, operations on partitions (i.e. allocation, garbage collection, etc.) can be organised in a two-dimensional array, indexed by the regime and operation (see figure 3.2.a). This is similar to the single-dispatching operation used in object-oriented languages to invoke a method on a given object [Ing86]. Maintaining this operation matrix, illustrated in figure 3.2.a, allows the Sphere developers to easily experiment and change the implementation of some partition operations, without impacting large parts of the Sphere code base. Additionally, it also provides a straightforward way of introducing new regimes into an already-existing store.

Figure 3.3 shows how an operation on a partition is invoked. Each partition contains an associated tag that determines its regime. This tag serves as an index into the two-dimensional operations matrix and, along with the operation index, yields the code for the desired operation. Then the code is executed, accepting the partition identifier and any other necessary arguments.

The Mneme object store [Mos90a] established a notion similar to regimes. In Mneme, they are referred to as *pools* and can be managed independently, allowing object formats to vary, implementing different buffer management. They even provide greater flexibility since it is up to the pool implementor to define their internal structure. This is not the case for the partition regimes of Sphere, since they have to conform to the structure described in section 3.6.2. This decision was taken as a compromise between flexibility and ease of implementation of new regimes.

3.5.2 Object Kinds

In PJama Classic, objects are divided into five categories: instances, class objects, scalar arrays, reference arrays, and bytecodes[8]. These categories, apart from the last one, directly correspond to Java data types and each of them has a different internal structure. Most similar systems use a number of distinct object types, especially if they are mappings of programming language types[9]. A category of objects of the same internal structure will be referred to as an *Object Kind* or just *Kind.*

[8]These are the byte arrays holding the results of compiling methods to bytecoded instruction sequences.

[9]There are also systems that have a uniform object format. Such a system is Napier88 [MCC+99, CCM99], where each object has all its reference fields grouped at the beginning, while a field on its header denotes how many of them there are [BR91, BM92, Bro89]. This object format was previously used in the PS-algol [ACC82], Emerald [BHJL86], and S-algol [Mor79] systems. Even though very simple and convenient, such an object format cannot be easily adopted by all systems, especially ones that support inheritance (all systems mentioned previously do not).

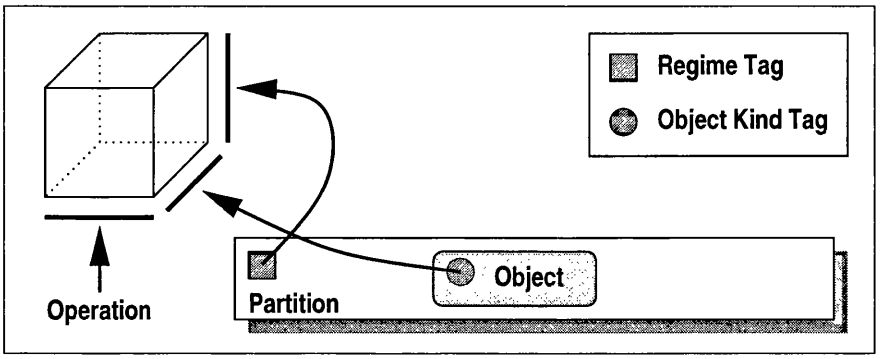


Figure 3.4: Invoking an Operation on an Object

There are several operations defined on objects, some being the same for all kinds (e.g. update single scalar field) and others requiring a different implementation for each kind (e.g. pointer identification). Further, it may be the case that some of these operations are regime-specific (e.g. fetching for small instances vs. large instances). So, in a similar manner to operations on partitions, operations on objects can be organised in a three-dimensional array, indexed by the object kind, regime, and operation, as illustrated in figure 3.2.b. Again, from an object-oriented point of view, this is a simple implementation of operation double-dispatching [Ing86].

The advantages of organising the operations on objects in such a matrix are similar to the ones in the case of partition regimes: ease of experimentation, single point of change, and ease of introducing new object kinds to a pre-existing store. Object kinds reflect differences in the internal layout of objects and may also support differences in their management. A few examples are presented below.

- **Java Strings** — Strings in Java are actually made up of two objects: *(i)* a character array and *(ii)* an instance of the class `java.lang.String` that contains three fields: a pointer to the character array, an offset into the array, and a length [GJS96]. This imposes an excessive two-object overhead per string in the store. A string object kind can be envisaged which flattens such strings into a single object, when writing them to the store, and recreates them, when fetching them into memory. GemStone Inc. also considered a similar approach [Oti98].

- **Compressed Objects** — It might be beneficial for large scalar arrays to be compressed when moved to the store to save transfer time and disk space. Examples are images (JPEG, GIF, etc. [KL92]), sound samples (MP3), etc.

- **Distribution Proxies** — Such objects reference other objects in remote stores and they might be necessary to be specialised at the store-level in future versions of PJama for efficiency reasons [SA97, Spe99].

Operations on objects are invoked in a similar fashion to operations on partitions. The regime tag and operation index are still needed, only this time a kind tag is also required. This is contained in the object's header and will serve as the third index in the three-dimensional operation matrix, as illustrated in figure 3.4. Once the code has been retrieved, it is executed accepting the object's PID and any additional arguments.

The object operations that are part of the Sphere public API are enumerated in section A.3.

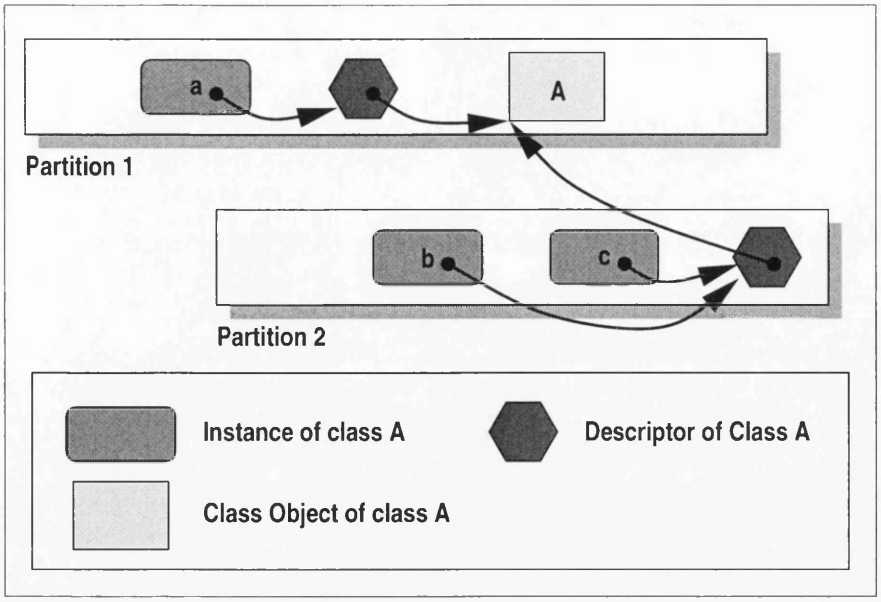


Figure 3.5: Concrete Example of the Use of Descriptors

3.5.3 Descriptors

Objects whose structure allows reference fields to be interleaved with scalar fields in arbitrary ways require a uniform way of specifying this information to Sphere. A *Descriptor* is a special object that contains information about the structure (at least the position of reference fields) of all objects with the same internal structure (i.e. all instances of the same class). Not all kinds of objects need a descriptor; e.g. scalar arrays do not need one (there are no references fields in them) nor do reference arrays (all their entries are reference fields). However, instances do. Notice that objects of the same kind might require different descriptors (e.g. instances of different classes will normally have different layouts). However, objects of the same internal structure (e.g. instances of the same class) can point to the same descriptor.

The use of descriptors is illustrated in figure 3.5 with a concrete example. All instances of class **A** (e.g. **a**, **b**, and **c**) point to the descriptor of class **A** that describes the layout of such instances. This descriptor points to the class object itself. This is necessary, since instances must reference their class objects. Since instances reference their descriptor, it is more space-efficient to hold the reference to the class in the descriptor. Notice that the descriptor of class **A** is replicated inside each partition which contains at least one instance of **A**. This keeps the descriptors "close" to the instances and minimises access to other parts of the store during garbage collection.

The main piece of information that needs to be included in a descriptor is the location of reference fields inside the corresponding objects. This can be represented in various ways: a bitmap with one bit per field in the object, a series of offsets of the reference fields, etc. (currently Sphere is using a bitmap). This can be augmented with information on the type of each field so that byte-order changes can be performed, when running on a machine of different endianness.

Each partition that contains objects that require a descriptor needs to manage a table of the descriptors allocated in it. Upon allocation of a new object, a look-up on this table will efficiently determine whether the corresponding descriptor already exists or needs to be allocated. A hash table is an appropriate data

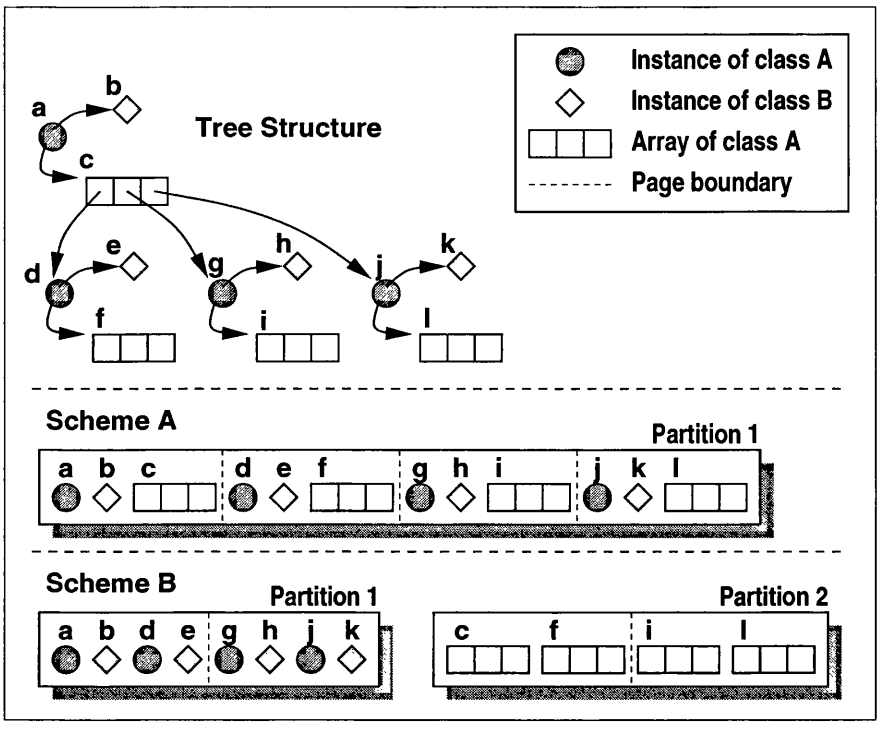


Figure 3.6: Concrete Example of Clustering Issues

structure for this (⇝§4.11.1). The presence of such a table can also allow the schema evolution mechanism to operate more efficiently, since it can quickly determine whether or not objects of a specific type reside in a given partition (⇝§6.7).

Finally, considering the concrete example in figure 3.5, it is apparent that the presence of descriptors also contributes towards minimising the cross-partition references from the instances to the class object to at most one per partition. This is beneficial since such references need to be recorded (⇝§3.6.2 and §4.10) for the correct operation of the garbage collector (⇝§6).

3.5.4 Clustering Considerations

It might seem that grouping objects in different partitions according to their kind, as mentioned above, would cause a high degree of declustering and hence penalise the performance of Sphere. However, this is not necessarily the case. Large data structures, which typically need to be clustered together (linked lists, trees, etc.), tend to be constructed from only a few distinct types of object, usually instances of a few classes and arrays. Hence, even though the instances and arrays will be stored in different partitions, as long as they are clustered close to each other within these partitions, the overall impact on performance will be low. It has also been observed that such data structures are usually larger in persistent systems than in traditional ones [AJ99a].

A concrete example follows. Observe the tree structure shown in the top section of figure 3.6 and consider how it may be copied to the store. Two copying schemes, illustrated in the second and third section of the figure, will be considered in turn.

According to scheme **A**, all objects are clustered in the same partition, irrespective of their kind. This keeps them close together and minimises disk accesses when the tree is traversed (depth first, left to right). However, object management within the partition may be more difficult and less efficient, since it has to deal with objects of different structure, size, and behaviour.

Alternatively, according to scheme **B**, instances are separated from arrays, when copied to the store. However, objects of both kinds will be clustered close to each other within each partition. Object management within the partition is now more efficient because it only has to deal with objects of the same kind. Initially, when the tree is traversed, pages from both partitions have to be accessed, making the startup cost more expensive than in scheme **A**. However, assuming that the entire tree structure is big enough not to fit in a single page, this cost will be absorbed as the rest of the tree is traversed and more pages are accessed.

In the example in figure 3.6, when the first node of the tree, containing objects **a**, **b**, and **c**, is accessed, scheme **A** will touch one page and scheme **B** two pages. However, when the next node, containing objects **d**, **e**, and **f**, is accessed, scheme **A** will touch a new page, whereas scheme **B** will touch the same two pages it previously touched that are very likely to still be in the disk cache. Therefore, the initial cost of touching two pages has already been absorbed. Obviously, this is a very specific example and the performance impact of either scheme is very application-dependent. However, there will always be pathological cases for both of them.

Finally, it is worth pointing out that storing partitions that hold different kinds (instances and arrays, in the case of the concrete example) on different physical drives has the potential to further improve performance, since it could minimise latency due to head movement.

Wilson has also reported gains when clustering objects of the same kind together [Wil97].

3.5.5 Optimised Dataflow

There are several ways to arrange the flow of data between the POS and the mutator. Figure 3.7 illustrates three of them. These are discussed below.

- **Scheme A** — It assumes that the store has been written specifically for the given mutator, therefore the mutator accesses it directly. This offers the highest *potential* performance. However, the store code is not generic and it is very prone to change, if the specification of the mutator changes.
- **Scheme B** — The store layer is general-purpose and totally independent of the mutator. However, since it is very likely that the object format it supports is different from the one the mutator uses, an extra translation layer is introduced to cope with this. This has a negative impact on the performance of the system. However, the store layer code is totally independent from the layer above it and can be easily re-used with only the translation layer having to be rewritten, if the specification of the mutator changes.
- **Scheme C** — This is the one which has been adopted in Sphere and has been proposed as a compromise between schemes **A** and **B**. The core of the store layer is generic, with only a set of well-specified operations (which define, among other things, the object format) having to be implemented specifically for the given mutator that uses the store. This way, only minimal translation intervenes and the store can be adapted to and optimised for particular situations. However, the use of the store is

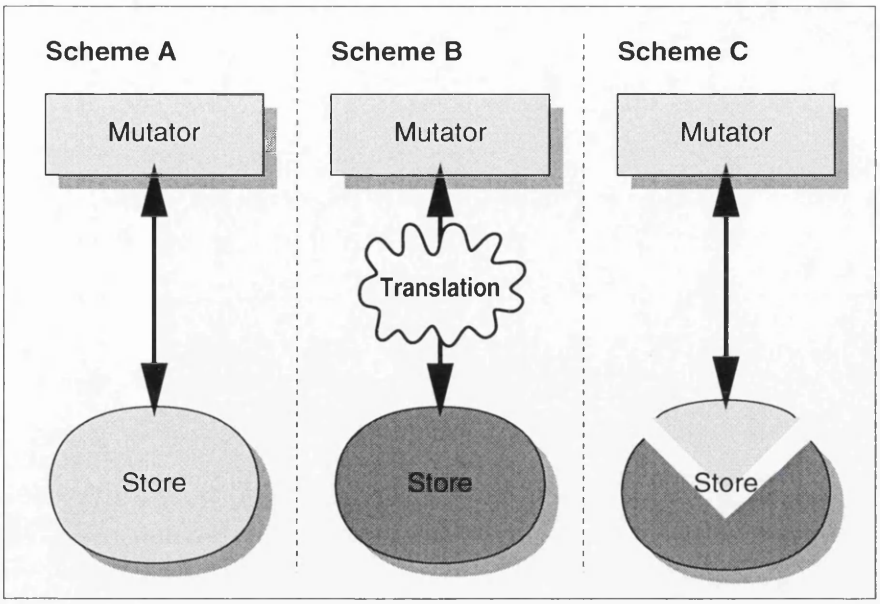


Figure 3.7: Dataflow Schemes between the Mutator and the Store Layer

not trivial, since the mutator implementer has to write the plug-in operations contained in the two operation matrices described in sections 3.5.1 and 3.5.2.

3.6 Partition Management

This section briefly describes the management of partitions. Section 3.6.1 discusses the partition identifiers and section 3.6.2 presents the partition layout.

3.6.1 Partition Identifiers

When a partition is created, it is allocated a unique identifier that it retains until it becomes empty and is reclaimed (if this ever happens). This identifier will be referred to as the *Partition Identifier* (PI). Logical, rather than physical, addressing is used for the PIs. This means that the PI of a partition is independent of its position within the store and does not change if the partition is moved, resized, garbage collected, etc.

One way to achieve this logical addressing is to introduce one level of indirection. It was decided to include a table, called the *Partition Table* (PT), in each Sphere store. Its entries correspond to allocated partitions and contain their physical locations. For efficiency, the PI of a partition is the index of its entry in the PT. Every time a partition needs to be accessed, its location is looked up in the PT. If the partition is moved, its location is updated in its PT entry without its PI needing to change.

More information on the implementation and management of the PT is given in section 4.8.

3.6.2 Partition Layout

Figure 3.8 illustrates the layout of a partition. The exact pattern of allocation will depend on the free-space management adopted by the regime managing the partition (⇝§3.5.1). However, for every object allocated,

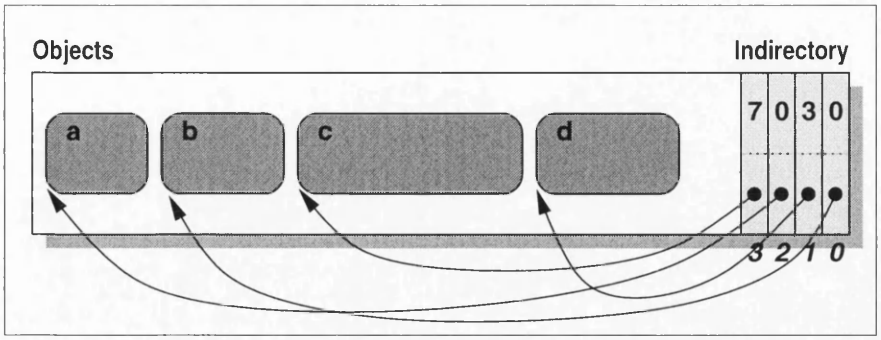


Figure 3.8: Layout of Partitions

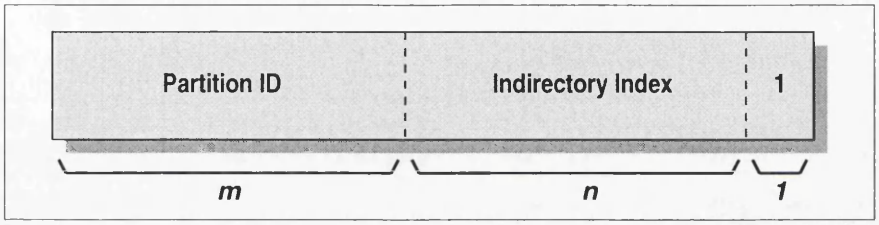


Figure 3.9: Sphere Persistent Identifier Format

a corresponding *Indirectory Entry* is also allocated. This provides one level of indirection to the object and its position inside the partition remains the same for the object's lifetime. Additionally, it also contains the object's reference count, used by the partition garbage collector (⇝§4.10 and §6). All indirectory entries are of the same size and are grouped at the end of the partition; this space is called the *Indirectory*.

3.7 Representing Persistent References

This section describes the logical addressing model that has been adopted in Sphere. Section 3.7.1 describes the format of the Sphere persistent identifiers, section 3.7.2 presents the logical view of the store presented to the mutator, and section 3.7.3 discusses a proposal on how to move objects from one partition to another.

3.7.1 Persistent Identifiers

Figure 3.9 illustrates the format of the Sphere PIDs. The least-significant bit of a PID is always 1 to distinguish it from memory addresses that are typically 4-byte or 8-byte aligned [Bro89, DA97]. The remaining space is split between the PI, which is independent of the partition location within the store, and the index of the indirectory entry of the object, which is also independent of the position of the object within the partition.

The above format can be adopted for both 32-bit and 64-bit PIDs. Currently Sphere uses the former. The reason for this is to use the same PID size as the pointer size used by the mutator (e.g. PEVM, ⇝§2.5). This allows the movement of objects between memory and disk (i.e. swizzling and unswizzling [WK92]) to take place efficiently, as the object size remains the same.

It can be proved that 32-bit PIDs, using the above layout, are more than enough to accommodate the Sphere target store size of 10GB. Assuming that the minimum mutator object size is 8 bytes, the minimum ob-

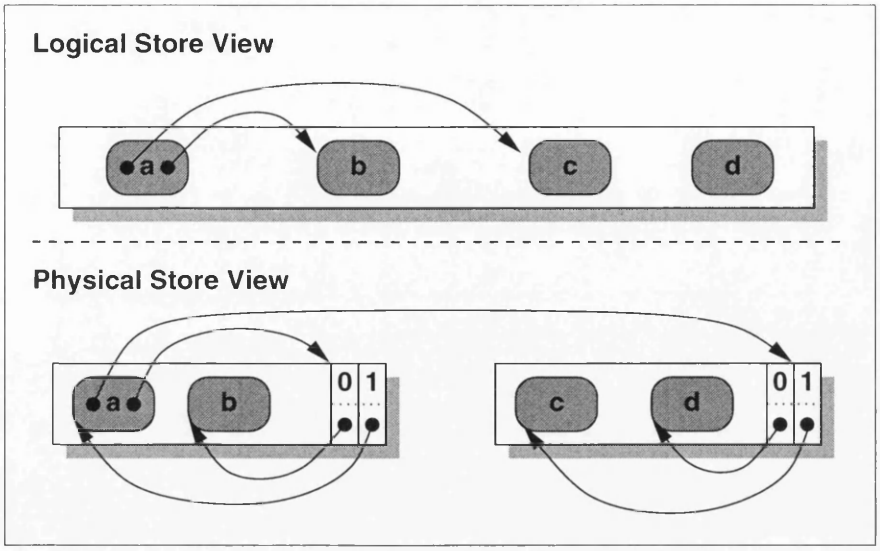


Figure 3.10: Logical and Physical Store Views

ject size in the store, including 8 bytes of header information (⇝§4.5.4) and 8 bytes for the indirectory entry (⇝§3.6.2), is 24 bytes. Assuming that the parameters in figure 3.9 are $m = 16$ and $n = 15$ (this is one possible combination but it might not be the optimal one), the minimum size of a full partition is $24 * 2^n = 3 * 2^3 * 2^{15} = 0.75$MB (roughly, ignoring partition header, free indirectory entries etc.). It follows that the minimum size of a full store is $0.75 * 2^m = 0.75 * 2^{16} = 48$GB with $1/3 = 16$GB of useful data (excluding header and indirectory sizes). However, this is the worst case and, if there are some large object partitions, this size will increase considerably and the ratio of useful data over total data will be much better.

It is possible for PIDs to be exhausted within a partition, without the partition being full. This happens when 2^n objects have been allocated in the partition without the object space having reached the indirectory space. In this case, the partition is considered to be full and, during the next garbage collection, an attempt will be made to decrease its overall size. Similarly, if the disk garbage collector detects that the partition is full but the PID availability is not, it will attempt an expansion.

3.7.2 Logical Store View

Figure 3.10 illustrates the logical view of the store that is presented to the mutator. Upon object allocation, it is up to the mutator to specify the regime under which the object will be stored. After that, Sphere will automatically choose an appropriate partition to allocate it, manage any necessary reference count updates, move the object inside the partition during garbage collection, de-allocate it, when it is not reachable, etc. All these operations take place transparently from the mutator, without exposing to it the Sphere internal data structures.

An important issue, illustrated in figure 3.10, is the reference count management. It is assumed that this particular store only contains the two partitions and the four objects shown. Object **a** is the persistent root, which is by default reachable and cannot be reclaimed. Hence, its reference count is 1, even though it is not reachable by other objects. Object **a** points to two other objects: **b** (this is an intra-partition reference, since both objects are in the same partition, and the reference count of **b** has not been increased) and **c** (this is a cross-partition reference, since the two objects are in different partitions, and the reference count of **c** has

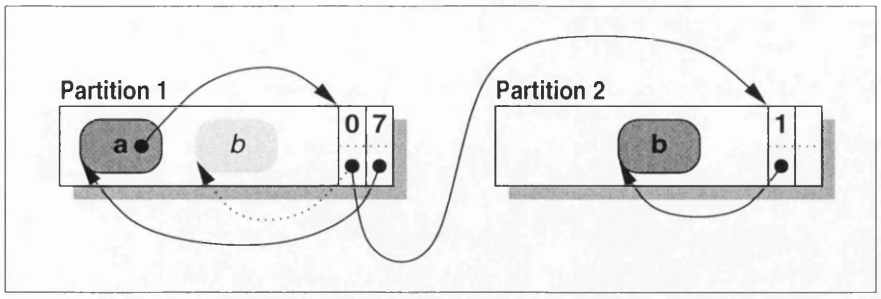


Figure 3.11: Forwarding References

been increased to 1). The only object which is not reachable and can be reclaimed is **d**, since its reference count is 0 and there are no intra-partition references to it.

In theory, it could be possible to optimise an intra-partition reference to point directly to the object, rather than to its indirectory entry, since *(i)* it does not affect the object's reference counts and *(ii)* the indirectory will not need to be visited, avoiding a potential disk access. There are two reasons why this has not be done. The most important one is that by pointing directly to the object, it is not easy to deduce its PID, since this requires the index of its indirectory entry (⇝§3.7.1) and there is no efficient way to retrieve it from the offset of the object inside the partition. The second reason is simplicity of the implementation. If all references are logical PIDs, upon compaction only the indirectory entries need to be updated, rather than all the intra-partition references in every object. This can accelerate significantly the compacting phase of the disk garbage collector, especially in highly inter-connected partitions [Pri96, Ham97].

3.7.3 Forwarding References

Sometimes, objects need to be migrated from one partition to another (e.g. when a partition is split, when two partitions are merged, or when an object increases its size so much its regime changes during evolution, ⇝§6.7). This introduces the complication that the object changes PID (since the PID depends on the PI) and any objects containing the old PID have to be updated. Scanning the entire store to identify such objects is infeasible and keeping remembered sets of the objects that point to each object will be too space-consuming and expensive to maintain.

A more efficient way to deal with this is to introduce *Forwarding References* (FRs). When an object is migrated to a new partition, the offset field of its indirectory entry in the old partition is overwritten with the new PID of the object. This is possible since it is straightforward to distinguish an offset from a PID (the least-significant bit is 0 or 1 respectively). So, when faulting an object given its PID, the indirectory entry is checked and if it is an offset, the object is retrieved, otherwise the look-up continues with the new PID. Chains of FRs can be allowed, if necessary.

It is believed that the number of FRs inside a store will be low since the events that create them (partition merge, partition split, etc.) are likely to be rare. Still, to eliminate performance problems imposed by their presence, it is necessary to eliminate them whenever possible. This can be done during garbage collection. When a partition is garbage collected, the objects in it are scanned and whenever a PID to a FR is detected, it is replaced by the new PID of the object it points to. This, however, will increase the number of page-faults occurring during garbage collection, since indirectories of other partitions will need to be accessed.

The use of FRs is illustrated in figure 3.11 in which object **b** is migrated from partition 1 to partition 2, while object **a** still points to the old indirectory entry of **b** in partition 1. The next time the garbage collector operates over partition 1, it will detect that a field in **a** holds a PID to a FR and will overwrite it with the new PID of **b**. Eventually, the FR in partition 1 will be reclaimed once it is not referenced by other objects.

If FRs are used, care has to be taken to manage their reference counts properly. Any references to an indirectory entry that contains an FR will increase the reference count of that entry by one. The FR itself will increment the reference count of the new indirectory entry of the object. When the FR is not reachable by any objects, it will be deleted, decrementing the reference count of the new indirectory entry, allowing the object to be collected, if necessary. Reference counts of chains of FRs will be handled in a similar fashion.

This management of reference counts is also illustrated in figure 3.11. The FR of object **b** from partition 1 to partition 2 has caused the reference count of **b** in partition 2 to be incremented by one. When partition 1 is garbage collected, the reference in **a** will be updated to point to **b** in partition 2, which will cause the new indirectory entry of **b** in partition 2 to be increased by 1 (it is now 2) and might cause the old indirectory entry of **b** in partition 1 to be unreachable. If the latter happens, the indirectory entry containing the FR will be garbage collected and this will cause the reference count of the new indirectory entry in partition 2 to be decreased (it is now back to 1).

The introduction of FRs has the potential to provide greater flexibility in Sphere. However, it will also complicate the object equality operation of the mutator, since it will allow two or more different PIDs to represent the same object. This can be dealt with in two different ways.

- When objects are fetched from the store into memory, all their reference fields are followed and, if they point to FRs, the "real" PIDs are substituted. This ensures that no PIDs that point to FRs will be visible to the mutator, hence the equality operation is unaffected. However, this will also cause unnecessary disk traffic during object-fetching and will have a negative effect on the performance of Sphere.
- Objects are fetched with no further checks. However, the object equality operation is altered to check, if necessary, whether the PIDs of the objects point to FRs and, eventually, to the same object. This has the advantage that only the object equality operation is affected. However, extra interaction with Sphere has to take place during the object equality operation.

The latter solution is probably more efficient, provided that it does not affect considerably the operation of the mutator. However, there are no plans to introduce FRs in Sphere as there are not immediate requirements for them.

3.8 A Two-Level Free-Space Management

This section describes the two-level free-space management model adopted in Sphere. Section 3.8.1 gives a brief overview of free-space management techniques, section 3.8.2 discusses partition allocation inside segments, and section 3.8.3 discusses object allocation inside partitions.

3.8.1 Free-Space Management Techniques

Free-space management deals with the management of a potentially large resource (e.g. disk space or main-memory heap). It keeps track of which parts of it (let's call such parts *chunks*) are not in use, it uses such

chunks to satisfy allocation requests, and it flags chunks as re-usable to satisfy de-allocation requests. There is a large literature dealing with free-space management and it will be impossible to include it here in its entirety. Instead, the reader is referred to an excellent survey by Wilson *et al.* [WJNB95] that outlines and compares different approaches in the context of main-memory heaps. Related issues are also discussed by Tannenbaum, in the context of operating systems [Tan90].

When designing the free-space management of a given resource, there are two main approaches that can be taken.

- **Compaction** — All the free space of the resource is grouped in one place and chunks are allocated from it linearly. De-allocation takes place by *compacting* all the used chunks, which yields a single contiguous free chunk. Even though this approach does not have the fragmentation problems that plague the in-place de-allocation schemes, as discussed below, it is not applicable to all situations, as chunks often cannot be moved. Compaction is usually associated with garbage collection, where no individual de-allocation requests take place; instead large numbers of chunks need to be de-allocated at the same time [App88, Wil92, Jon96].

- **In-Place De-allocation** — A record of the location of free chunks within the resource is kept. When an allocation request needs to be satisfied, an appropriate free chunk is discovered and marked as used. When a de-allocation request is raised, the corresponding chunk is marked as re-usable and added to the set of free chunks. Notice that de-allocations do not require the used chunks to be relocated.

  Unfortunately, the in-place de-allocation approach can introduce *fragmentation*. This happens when an allocation request cannot be satisfied, since all the available free chunks are smaller than the requested size, even though their total size might be larger [WJNB95]. Obviously, fragmentation is an important problem, as an allocation request cannot be satisfied, even though enough free space is available, and care has to be taken by the allocation mechanism to attempt to reduce it [JW98].

For the in-place de-allocation approach, several data structures have been proposed to manage free-space, providing different trade-offs between time and space efficiency and fragmentation reduction. Some are outlined below.

- **Free-Lists** — The free chunks are organised in a linked list structure. When an allocation request needs to be satisfied, this list is traversed until an appropriate chunk is found. Free-list implementations attempt to coalesce adjacent free chunks. Several allocation policies have been proposed that attempt to reduce fragmentation: *first-fit*, *best-fit*, *next-fit*, etc. [WJNB95].

- **Size-Segregated Free-Lists** — As above, but several free-lists are used, each of which contains free chunks with a different range of sizes in an attempt to minimise scanning time [WJNB95].

- **Buddy Systems** — Buddy systems [Kno65, PN77] are a variant of segregated free-lists, supporting a limited but efficient kind of splitting and coalescing. In simple buddy systems, the entire resource is conceptually split into two large areas which are called *buddies*. There areas are repeatedly split into two smaller buddies, until a sufficiently small chunk is achieved. This hierarchical division of the resource is used to constrain where used chunks are allocated and how they may be coalesced into larger free chunks. Several optimisations on this scheme have been proposed: *binary buddy* [Kno65], *double buddy* [Wis78], etc.

- **Bitmaps** — The resource is split into equal-sized chunks and a bitmap is maintained, containing one bit per such chunk. A chunk is free, iff its corresponding bit is 0; or used, iff its corresponding bit is

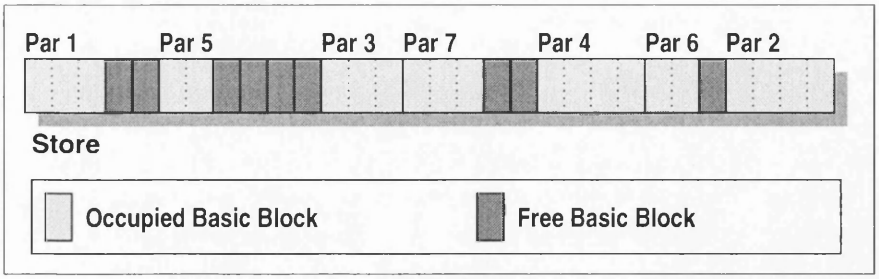


Figure 3.12: The Use of BBs

1. When an allocation request needs to be satisfied, the bitmap is scanned until a series of 0s, long enough to satisfy the given allocation request, is discovered. Even though bitmaps are not used very extensively in the context of memory management, they are very appropriate for this task since *(i)* they can be scanned efficiently, one 4-byte or 8-byte word at a time [Pri96], and *(ii)* they also provide information on the used chunks as well as the free ones, allowing the allocation mechanism to be able to reason about the location of the chunk to be allocated [Wil97, Joh97].

3.8.2 Allocating Partitions

In most cases, compacting an entire persistent store is impractical, due to its sheer size and the time that such an operation will take. So, it was decided that an in-place allocation/de-allocation scheme was to be adopted for partitions in Sphere. Each store segment is split into fixed-size blocks, called *Basic Blocks* (BBs)[10]. The BB size is currently set to 0.5MB. When a partition needs to be allocated in a store, a segment is sought that contains a number of contiguous BBs, equal to the requested size for the partition. These are allocated for the partition. It follows that partition sizes can only be a multiple of the BB size. When a partition is empty and needs to be de-allocated, the BBs it occupies are marked as free in order to be re-used later. However, and unlike typical in-place allocators, partitions in Sphere can also be moved for garbage collection purposes. This is achieved by reserving the BBs at the new location of the partition and marking as free the old ones, once the partition has been moved. The use of BBs is illustrated in figure 3.12.

The data structure that was chosen to manage free BBs is a bitmap. Each Sphere segment has such a bitmap, with one bit for each BB it contains. Apart from the reasons presented above, bitmaps were chosen also because of the following.

- **Compactness** — Given a BB size of 0.5MB and a maximum segment size of 2GB, a segment can contain a maximum of 2GB/0.5MB = 4096BBs. The corresponding bitmap will be of size 4096/8 = 512 bytes (given that each byte on the bitmap can represent 8 BBs), which fits entirely on a store page. In fact, the pages containing these bitmaps (one page per store segment) can be pinned in memory, to avoid them being evicted. This way, BB allocation can take place with no I/O cost.
- **Efficient Updates** — Allocating/de-allocating BBs using a bitmap can be done efficiently by flipping the corresponding bits. However, most importantly, the log traffic that is generated to ensure the fault-tolerance of the operation (⇝§4.4.2) is minimal, comprising a single small log record that contains the indexes of the first and last bits that were flipped.

[10] A better name for them would have been *Minimum Blocks*, but unfortunately this is abbreviated to MBs, same as megabytes. Andrew Black proposed to use BBs instead.

3.8.3 Allocating Objects

As discussed earlier in this section, different free-space management techniques can be applied to partitions of different regimes, optimised for the kind of objects they contain.

- Small object regimes can benefit from compaction, as it can eliminate fragmentation in the corresponding partitions. Typically, such partitions are relatively small in size, hence compacting them is feasible [Pri96]. It turns out that the side-effect of compaction for creating a single contiguous free chunk also contributes to the efficient bulk allocation operation in such partitions (↝§5). Currently, compaction is the only free-space management technique implemented in Sphere.
- An in-place de-allocation scheme can be applied to large object regimes, to avoid excessive copying. A bitmap structure is appropriate for such partitions, for the reasons presented in section 3.8.2.

In a long-running Sphere store, the latter mechanism might eventually introduce fragmentation in partitions managed by it. However, given the logical addressing mechanism of Sphere (↝§3.7.1), compaction could be applied to such partitions to deal with this problem, as a last resort.

3.9 Summary

This chapter has presented the basic design of the Sphere POS. A brief description of the envisaged situations in which Sphere will operate (↝§3.1) and its design goals (↝§3.2) were given, as well as the reasons why an object-based approach was adopted (↝§3.3). Then, the container and abstraction hierarchies introduced in Sphere were described (↝§3.4 and §3.5). The management of partitions (↝§3.6) was then given, followed by the representation of persistent references (↝§3.7). Finally, the two-level free-space management of Sphere was discussed (↝§3.8).

The next chapter presents the implementation of Sphere, which was based on the design given here. It describes the main management structures of a Sphere store and implementation techniques that contribute towards its efficient and long-running operation.

He's more machine now than man; twisted and evil.
— ***Obi-Wan Kenobi***, *Jedi Master*

Chapter 4

The Implementation of Sphere

This chapter presents details on the implementation of Sphere (see chapter 3 for its high-level design). Section 4.1 presents the organisation and interaction of the main Sphere components and section 4.2 describes the two different types of locks used to ensure MT-safety. Section 4.3 describes two benchmarks that are used throughout this chapter to provide measurements of different aspects of the system. The recovery sub-system, which ensures fault-tolerance, is described in section 4.4 and the layout of the main entities in Sphere (segments, partitions, indirectories, and objects) is given in section 4.5. The management of BBs, with respect to their addressing scheme and allocation policy, is discussed in section 4.6. The implementation of disk cache, used for caching store pages, is described in section 4.7. The partition table, whose main role is to provide one level of indirection to the partitions, is described in section 4.8. Section 4.9 presents caching issues for objects and partitions and section 4.10 deals with grouping reference count updates. Section 4.11 discusses the management of descriptors and section 4.12 summarises concurrency issues. Section 4.13 enumerates the customisation (kinds and regimes) implemented for the PEVM system (↝§2.5). How Sphere meets its requirements (presented in section 3.2) is discussed in section 4.14. Section 4.15 concludes the chapter.

A slightly different version of this chapter has been published as a technical report [PAD98].

4.1 Component Organisation

Figure 4.1 illustrates the main components of Sphere and how they interact with each other. Great detail on their interaction is not given in the figure for the sake of clarity. The thick black lines are the interfaces between components and the thick grey lines represent the public Sphere API (↝§A). The main five components are described below, along with their contents.

- **SphereLib** — In order to facilitate its porting to different platforms, all the operating system and architecture-dependent facilities that Sphere needs have been included in a single module called SphereLib. Such facilities are I/O access, thread model, signal-trapping, etc. Additionally, some utilities that need to be shared between the Sphere core and the log manager are also included in SphereLib (parsing facilities, fingerprinting implementation, etc.).

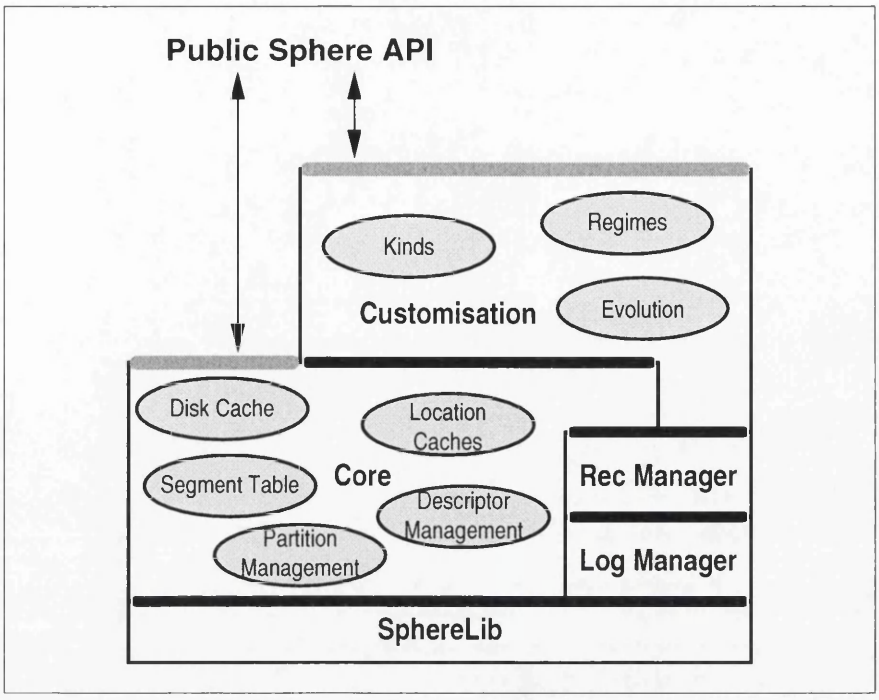


Figure 4.1: Component Organisation

- **Log Manager (⇝§4.4.2)** — A generic module that manages virtual logs and log sequence numbers. The log record contents are load-independent, therefore this module can be re-used, if necessary, in contexts other than Sphere.

- **Recovery Manager (⇝§4.4.3)** — This module can be seen as the Sphere-specific instantiation of the log manager. It is responsible for filling-in the contents of log records, interpreting them at recovery time, and applying the undo and redo operations that are necessary to eliminate inconsistencies in the store. Currently the recovery manager contains all necessary operations to support both the core and customisation parts of Sphere. In the future, some of its facilities might be moved to the customisation part, for better separation of the two components.

- **Store Core** — This module is the Sphere core, managing all its generic data structures. These can be considered as the "building blocks" the customisation operations use. The core also contains some of the public API calls that are independent of the customisation (i.e. store creation, opening, etc.). These are described in section A.2. Its five most important subcomponents are the following.

  - **Basic Block Management (⇝§4.6)** — It comprises facilities to allocate/de-allocate BBs, address BBs, and abstract over the existence of multiple store segments. It also manages the segment table.
  - **Disk Cache (⇝§4.7)** — It is responsible for caching store pages (⇝§3.4.2).
  - **Partition Management (⇝§4.8)** — It deals with the allocation/de-allocation of partitions and locking partitions for updates, garbage collection, etc. It also manages the partition table.
  - **Location Caches (⇝§4.9)** — Caching facilities for some aspects of Sphere's operation. These are also used for some additional locking facilities.

- **Descriptor Management (↝§4.11)** — It comprises the mechanisms that manage descriptors inside a partition and cache their location.

- **Store Customisation (↝§4.13)** — This module is the customisation part of Sphere. It is tied to the mutator that Sphere needs to serve (↝§3.5.5). Most of the public API calls are contained in this module. Its three most important subcomponents are the following.
  - **Regime Implementation (↝§4.13.2)** — The implementation of the regime operations, described in section 3.5.1. These include free-space management, garbage collection, etc.
  - **Kind Implementation (↝§4.13.1)** — The implementation of the kind operations, described in section 3.5.2. These include fetching, updates, reference-field discovery, etc. and constitute the main part of the Sphere public API, described in section A.3.
  - **Evolution (↝§6.7)** — This comprises the schema evolution facilities that are required to evolve the contents and structure of objects [HAD99].

4.2 Locks

Sphere uses two types of locks to ensure *multi-threaded safety* (MT-safety). Both of them ensure that only one thread can hold the lock at any time. However, there are some differences between them.

- **Mutex** — A heavy-duty lock that queues all threads that are waiting to take it to prevent thread starvation. It also supports recursive locking/unlocking by the same thread, if this is required. Currently, it is based on the mutex implementation that is provided with the thread package adopted in SphereLib (e.g. `pthread_mutex_t`, if using POSIX threads [LB98] — in this case the size of the mutex data structure is 24 bytes). It is mainly used for situations when the lock is held for long periods of time and for "hot" locks that are expected to be contended for by a large number of threads.

- **Latch** — A lightweight lock that is implemented using the atomic *Compare And Swap* (CAS) instruction[1] of the underlying architecture (e.g. the `cas` instruction on SPARC chips [SPA94]). No thread queueing is implemented (a thread keeps trying to take the latch until it is successful, sleeping for a small amount of time between tries to avoid busy-waiting) and threads might starve. However, taking/releasing a latch is very fast (typically, about three times faster than a mutex) and the latch data structure very small (a single 4-byte word). It is used in situations where the lock needs to be held for a very short period of time (typically, enough for a few pointer manipulations) and for locks that are likely not to be contended for by a large number of threads.

Both types of locks are used in the Sphere components, as described in the rest of this chapter. Additionally, the CAS instruction was also used to atomically increase and decrease counters. In this case, the thread reads the value of the counter, calculates the new one, and using the CAS instruction attempts to atomically install the new value, provided that the counter has not been updated in the meantime. If the value of the counter has changed, the thread repeats the process, *without* sleeping in the meantime (it is assumed that the operation will succeed soon as the critical region between reading the counter and performing the CAS instruction is very short and is unlikely, even though not impossible, that during it another thread will update the counter). The fast path of increasing a counter using this method is in practice about 1.5 times faster than the fast path of taking a latch, increasing the counter, and releasing the latch (assuming the above implementation of the latch). Also, CAS does not have the memory overhead of the latch (even though this is very small) and in some cases of high contention performs better in improving concurrency.

[1] This is also sometimes referred to as the *test-and-set* instruction.

4.3 Benchmarks

In order to demonstrate how some components of the system behave, two benchmarks were defined and used to obtain measurements for these components. These benchmarks concentrated on single-threaded and multi-threaded object-faulting (the operation that is expected to be the most common) and were intentionally kept very simple. The main reason behind this decision was that it is easier to understand a component's behaviour when using a simple and easy-to-understand program rather than a large and complex application (strictly, they should be referred to as microbenchmarks). It is worth pointing out that the measurements obtained from the two benchmarks and presented in later sections of this chapter are included only for illustration purposes, are not meant to be full-fledged experiments, and were not the only benchmarks used to measure and tune Sphere.

The Sphere store used in both benchmarks contained 500 disjoint linked lists and each list contained 1,000 24-byte nodes. The nodes of a list were placed in the store so they were interleaved with nodes from other lists (i.e. a single page contained nodes from several lists rather than just one). The store was 21MB in size and contained 20 1MB partitions and 500,000 objects. The configuration of Sphere was a 10 entry partition location cache (↝§4.9.2 — this was 50% of the partitions in the store) and a 5MB disk cache (↝§4.7 — this was 25% of the store size).

The two reader benchmarks were the following.

- **SingleBench** — Iterate over the 500 lists and time how long it takes to do so (less is better).
- **MultiBench** — Spawn 40 threads, each of which chooses a list at random and iterates over it. Run the benchmark for 5 mins and count the benchmark throughput, i.e. the total number of lists read in their entirety by all threads (more is better).

The store startup and shutdown times were not taken into account. Also, in order to be kept simple and predictable, neither benchmark performed any object-caching and always read an object from the store when it required it.

The benchmarks were run on a Sun Enterprise 450 server with four 300MHz UltraSPARC-II CPUs, an UltraSCSI disk controller, and 2GB of main memory [Sunwww]. The machine runs the Sun Solaris 7 operating system. The single store segment and the log resided on two separate but identical physical disks (9.1GB Fujitsu MAB3091, 7,200rpm [Fujwww]). Additionally, raw partitions were used instead of files in an attempt to avoid the results being skewed by the operating system disk caches [Ber98].

4.4 The Recovery Mechanism

This section briefly describes the recovery mechanism of Sphere that ensures fault-tolerance and guards the store against crashes. Section 4.4.1 introduces the notion of a history, the Sphere-equivalent of a transaction, section 4.4.2 describes the Sphere log manager, and section 4.4.3 the Sphere recovery manager.

The Sphere log manager and recovery manager are described in greater detail in two technical reports by Hamilton [Ham00, Ham99a].

4.4.1 Histories

A *History*[2] is the Sphere-equivalent of a transaction [GR93]. Any updates to the store have to take place in terms of a history. All updates of the same history will be applied to the store atomically, i.e. either in their entirety or not at all. This atomicity guarantees that the store is not left in an inconsistent state, following a store crash, power failure, etc. However, it is up to the mutator to decide which updates will take place in terms of which history. This ensures that no application-level inconsistencies can happen. This is because it is the mutator, and not the store, that can reason about the contents of the objects and the invariants imposed on them.

In order for the mutator to apply any updates to the store, it has to first create a new history (using the `sphere_createHistory` call, ⇝§A.2.8), apply any necessary updates in terms of it (e.g. object updates, new object allocations, and registering a new persistent root — notice that all the corresponding operations/calls accept a history as an argument), and finally commit it (using the `sphere_commitHistory` call, ⇝§A.2.9). If the commit operation completes successfully, Sphere guarantees that the updates that have just been committed will survive any subsequent failures (this does not include media failures on the disks on which the store segments are stored).

Currently, nested histories are not allowed, i.e. histories are flat, and no explicit rollbacks are supported. These facilities have not been considered yet as they have not been requested by the main Sphere "user", PEVM (⇝§2.5). However, concurrent histories are allowed, typically by "attaching" each of them to a different mutator thread.

Sphere maintains an in-memory table, called the *History Table*, that contains one entry for each *active* history, i.e. a history in terms of which updates are still taking place and which has not yet been committed. Each table entry contains information on the status of the history, its corresponding virtual log (⇝§4.4.2), and a reference to its reference count update buffer (⇝§4.10). The size of this table has been arbitrarily fixed to 24 entries. However this can be trivially increased to meet a mutator's needs. When a history is created by a mutator, an entry in the history table is allocated for it. The ID of that history is the index of its entry in the table. After the history has been committed, its corresponding entry is de-allocated and can be re-used to satisfy another history-creation request. It follows that, during Sphere's execution, the same history ID can be allocated for consecutive histories. However, such histories are considered and treated as fundamentally different, as explained in section 4.4.2. MT-safety on the history table is provided by a single global lock (a mutex), as operations that change it are expected to be relatively infrequent.

4.4.2 The Log Manager

To ensure the atomicity of updates, described in the previous section, Sphere depends on the presence of a *Log* [MHL+92, MHL+89, RM89]. The log resides on disk and comprises a series of *Log Records*. Each log record corresponds to an update in the store and each update has a corresponding log record. Sphere has adopted the widely used Write-Ahead Logging (WAL) scheme [MHL+89, MHL+92], according to which, before any dirty store pages are written to the store, the corresponding log records must be written and flushed to disk. This way, if a crash occurs while writing the store pages, the log is guaranteed to have enough information to be able to redo or undo the required changes in order to re-establish store consistency. The operation that, following a crash, reads the log and redoes/undoes updates to the store is called *Recovery*.

[2]The term history was chosen over transaction to avoid confusion between transactions at the store-level and transactions at the mutator-level.

Log records are always appended at the end of the log, to ensure that no information, which might be needed by the recovery operation, is overwritten. Unfortunately, physical disks of infinite capacity do not exist[3]. Therefore the log implementation, even though it has to present the illusion of an infinite log, needs to safely discard parts of it that are guaranteed not to be needed any more (because, say, the corresponding updates have been written to disk and have been committed). It has to do so in order to fit the essentially infinite log in a finite physical space. This operation is referred to as *Log Compaction.*

In Sphere, the component that manages the log is called the *Log Manager.* It is responsible for appending log records to the log and ensuring that they are flushed to disk, when a history-commit request is raised. It also allocates a *Log Sequence Number* (LSN) for each log record that it appends. An LSN is an identifier for each log record that can be used to address it. LSNs are unique for the entire lifetime of the log. Additionally, as log records are appended linearly to the log, the order of the LSNs reflects the chronological order of the corresponding updates. The use of LSNs, in the context of recovery, is described in section 4.4.3. Currently, LSNs are 64-bit words.

A collection of log records that correspond to updates that have to be written atomically to the store (e.g. all updates of the same history ⇝§4.4.1) is referred to as a *Virtual Log.* Each virtual log has a 32-bit ID that is unique for the entire lifetime of the log. When Sphere allocates a new history, a corresponding virtual log is also allocated inside the log manager. Every log record appended in terms of that history is "tagged" with the ID of that virtual log. In this way, all the log records of the same history have the same unique virtual log ID on disk and can be distinguished from log records of other histories (even if their history IDs in memory happened to be the same, ⇝§4.4.1).

As information is always appended to the log, log records of different virtual logs may be interleaved on disk. This happens if they are written concurrently, as no virtual log takes an exclusive write-lock on the entire log (concurrency control ensures that log record writes are serialised). All log records of the same virtual log are linked with both forward and backward links. In this way, iteration over them can be performed efficiently (⇝§4.4.3).

Finally, it must be emphasised that, as already mentioned in section 4.1, the implementation of the Sphere log manager is general-purpose and not Sphere-specific. Even though it manages virtual logs, allocates LSNs, and links the log records of the same virtual log, the contents of the log records themselves are unknown to and cannot be interpreted by the log manager. Hence, it can be adopted by other applications, if necessary.

4.4.3 The Recovery Manager

At the beginning of Sphere's execution, the store is checked to determine whether its previous execution was shutdown gracefully or was terminated abnormally by a crash or a failure. In the latter case, there is the possibility that not all updates were written to disk and the store is in an inconsistent state. It is the responsibility of the log manager to determine whether recovery is needed or not (by checking that all virtual logs had been terminated correctly). However, it is up to another Sphere component, namely the *Recovery Manager,* to interpret the log records that have been written to the log and apply the appropriate changes to

[3]Interestingly enough, the currently high capacity of disks on the market can be considered as "infinite" for some applications. In particular, Hamilton described the latest 51GB Seagate disk as *"almost write-only"* [Ham99b].

the store to eliminate any inconsistencies.

The recovery algorithm that has been implemented in Sphere is based on the well-known and widely used *ARIES algorithm* [MHL+92, MHL+89, RM89, Moh99]. The adoptation of ARIES has allowed Sphere to implement a *steal policy* on its disk cache, i.e. dirty pages can be evicted to disk at any time. This overcame the scalability problems experienced in the PJama Classic system, caused by the no-steal policy imposed by the use of RVM (as described in section 2.4.1). The invariant is that, before a dirty page can be written to disk, all the log records that correspond to the updates applied to it are forced to disk. This can be done by keeping track of the LSN of the last update performed on each page. Forcing the log up to and including that LSN guarantees the above assumption.

The recovery operation is performed in two stages. During the *redo stage*, the log is scanned forward and all logged updates that had not been written to the store before the failure are redone. The WAL policy guarantees that such updates had been written to the log and flushed to disk. The redo stage ensures that the store is brought exactly to the same state it was when the failure occurred. Then, the *undo stage* scans the log backwards and rolls back all uncommitted updates that had been written when dirty pages were evicted from the disk cache.

The original ARIES algorithm assumes that all disk pages contain an LSN corresponding to the last update that has been applied to them. This allows the recovery process to check whether an update that corresponds to a given log record has been applied on the page or not (if the LSN on the page is less than or equal to the LSN of the log record, the update has been applied, otherwise it has not). This allows the recovery operation to determine whether it needs to redo or undo an update. In Sphere, however, this assumption was relaxed and LSNs are only placed on pages that need them (i.e. the ones that can have logical updates, ⇝§4.4.3.1). In this way, objects and other data structures can be laid out more naturally and efficiently. However, the penalty of this approach is that some redundant work might take place during the recovery operation, affecting its performance. This trade-off was considered acceptable, as this operation only takes place following a crash, and the "normal" Sphere operation is left unaffected.

In Sphere there are two kinds of updates: *physical* and *logical*. These are described in sections 4.4.3.1 and 4.4.3.2 respectively, along with the way their corresponding redo and undo operations operate.

4.4.3.1 Physical Updates

Physical updates are idempotent to being redone and undone. For example, the update of an integer field from 3 to 5 is considered a physical update, since it can be redone (by setting it to 5) or undone (by setting it to 3) many times, with the same end result. The idempotent properties of such updates allow the recovery mechanism not to have to depend on the presence of an LSN on the page to determine whether to redo or undo them. It follows that any pages whose contents can only be altered by physical updates, do not require an LSN stored on them (the majority of pages in Sphere stores fall in this category).

To ensure the correctness of the redo/undo operations of physical updates, the corresponding log records contain the before- and after-image of the update (3 and 5 respectively, in the case of the above example). At recovery time, a physical update is always redone or undone (this depends on the phase of the recovery operation) by writing the after- or before-image respectively, even if the corresponding page had been written to disk before the failure. This is safely done based on the idempotence of physical updates, as described above.

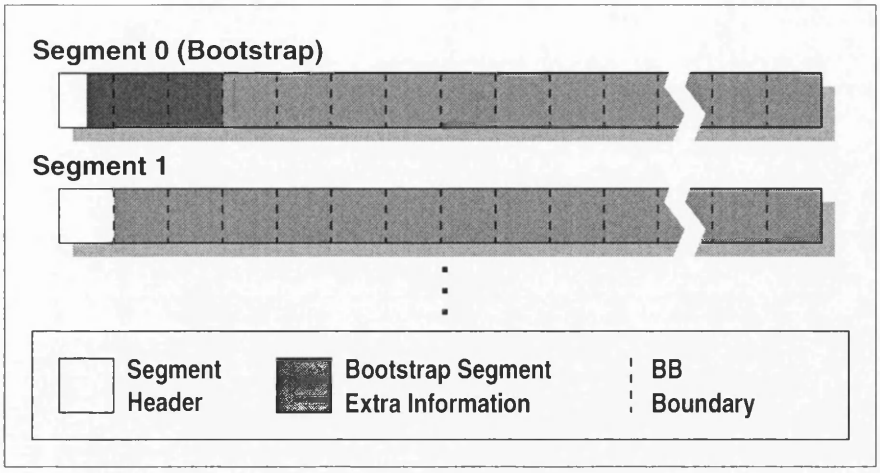


Figure 4.2: Layout of Segments

4.4.3.2 Logical Updates

Contrary to physical ones, logical updates are not idempotent to being redone and undone. For example, increasing a reference count by one is considered a logical update since, if it is applied twice, the resulting value will be wrong (one more than it should be). To ensure that such updates are redone or undone only when it is necessary, pages whose contents can be altered by logical updates have to contain an LSN corresponding to the most recent update applied on them. This ensures that during the redo operation only the logical updates that have not been stored on disk are redone and during the undo operation the ones that have been stored on disk are undone.

The log records corresponding to logical updates do not need to contain the before- and after-image of the update, unlike the ones for the physical updates. Instead, they contain an abstract description of the update, along with enough information to redo or undo it. For example, in the case of the reference count increase example, the corresponding log record will contain a description to "increase the reference count by one", instead of, say, 5 and 6.

4.5 Layout of Disk Data Structures

This section describes the layout of the main Sphere structures. In particular, the layout of segments is presented (⇝§4.5.1), as well as that of partitions (⇝§4.5.2), indirectories (⇝§4.5.3), and objects (⇝§4.5.4).

4.5.1 Segment Layout

Figure 4.2 illustrates the layout of Sphere segments. Each segment is split into a number of BBs (⇝§3.8.2) and its size is a multiple of the BB size.

The first BB of each segment includes the segment header that contains information on the store identity (⇝§4.6.1), as well as the allocation bitmap for that segment (⇝§4.6.3). Even though the information included in this header is considerable smaller than the size of a BB, it was decided that an entire BB be allocated for it. This kept the management of the segments simpler and provided space for any potential future header expansion that might be necessary. Additionally, it does not pose a performance problem, as

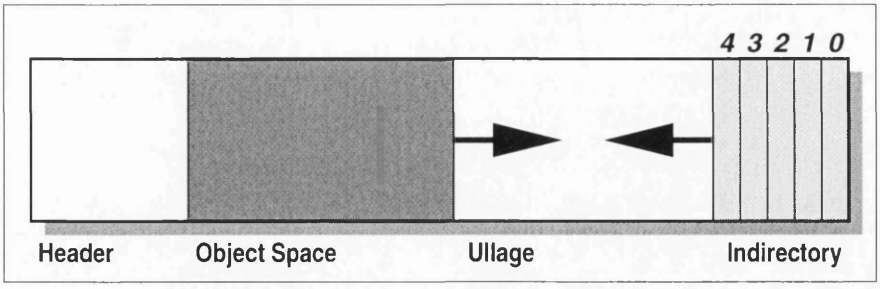


Figure 4.3: Layout of Partitions

the empty space at the end of the first BB is not accessed during normal Sphere operation.

An exception to the above is segment 0, the *Bootstrap Segment*, which is treated specially, as it contains some extra information, mainly the PT (⇝§4.8). This resides at the end of the standard segment header and is extended into the next few BBs (currently, two extra BBs are needed).

4.5.2 Partition Layout

Figure 4.3 illustrates the layout of the components of a partition. These are described below.

- **Header** — where management information for the partition, such as free-space management, descriptor table (⇝§3.5.3), etc. is stored. For simplicity reasons, and to be able to accommodate future expansion, the first page of a partition is entirely dedicated to its header. Typically, the partition header is not accessed during object-faulting, but only during object allocation (i.e. promotion, ⇝§5).
- **Object Space** — where objects are allocated. As its size increases, the object space grows forward in the partition.
- **Indirectory** — where indirectory entries, which provide one level of indirection to the objects, are stored. Indirectory entries are described in more detail in section 4.5.3. As its size increases, the indirectory grows backwards in the partition.
- **Ullage**[4] — free space of the partition into which both the object space and the indirectory can grow.

Notice that the partition layout is similar to earlier work by Atkinson *et al.* on the CMS system [ACC83b].

A partition is considered full when: *(i)* the object space and the indirectory cannot grow further[5], *(ii)* the PIDs corresponding to that partition have been exhausted (⇝§3.7.1), or *(iii)* there is no more space to allocate a new descriptor needed by an object.

4.5.3 Indirectory Layout

Each allocated object inside a partition has a corresponding indirectory entry. This provides one level of indirection to the object. An indirectory entry contains the following two fields.

[4]This term has been borrowed from wine making, bulk cargo-handling, and liquid fuel rocketry, where it is used to mean "space for expansion" above the relevant liquid.

[5]In this case, the partition garbage collector (⇝§6.3) will be able to expand the partition size so that more objects can be accommodated in it.

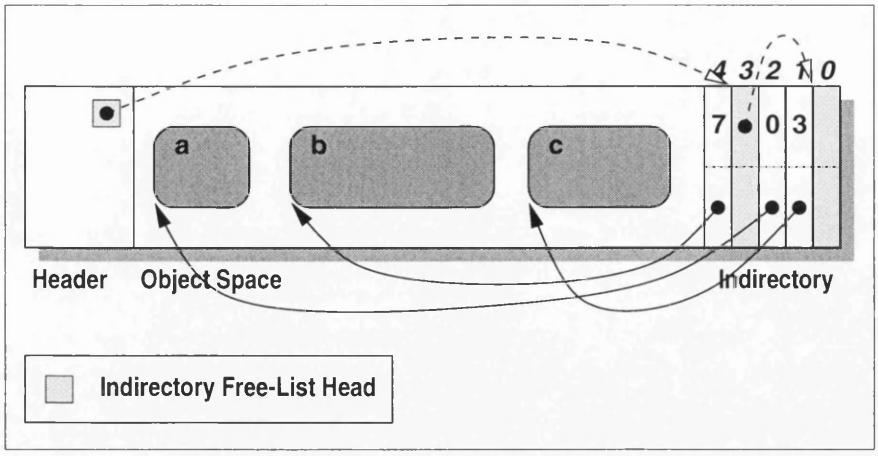


Figure 4.4: Organisation of Indirectories

- **Object Offset** — The offset of the corresponding object inside the partition from the start of the partition. This is independent of the position of the partition within the store.

  It is assumed that an unsigned 4-byte word will be sufficient for this, since partitions will never be as large as 4GB. In fact, it might be possible to allocate some of the bits in this word for other uses (e.g. various flags). However, if forwarding references to other partitions are allowed (⇝§3.7.3), it will be necessary to accommodate PIDs in this space, therefore all the 32 bits will be needed.

- **Reference Count** — The number of references to the corresponding object from objects in *other* partitions. These are used by the partition garbage collector (⇝§6).

  It is assumed that an unsigned 4-byte word will be sufficient to accommodate a reference count, since it is virtually impossible that there will be more than 4 billion cross-partition references to a single object in a single 10GB store.

An alternative scheme would have been to allocate only 2 bytes for the reference counts. This increases the probability of overflow, in the pathological case, when there is a large number of references to a single object. However, if this happens, this object can be considered to be live for the remaining lifetime of the store (since it is so heavily referenced, it is very likely that it will remain live). This scheme allows for the other 2 bytes of the second field to be used for something else. For example, they can be used to store a reference to the descriptor (⇝§3.5.3), assuming that special structures have been setup in order to identify descriptors within a partition given only a 2-byte ID; this could eliminate the need for an extra 4-byte word to accommodate the description PID.

When an indirectory entry is allocated for an object, it keeps the same position inside the partition during the entire lifetime of that object. If the object is moved inside the partition, the object offset of its indirectory entry is updated appropriately.

Indirectory entries which have been freed (when their corresponding objects have been reclaimed) are linked together in a list called the *Indirectory Free-List*. The indirectory will grow only when this list is empty. It can also shrink, if a number of contiguous entries at its end have been freed.

Figure 4.4 gives a concrete example of the operation of the indirectory. Entries **0** and **3** have been freed, appear in the free-list, and will be allocated to the next two objects allocated to this partition. If a third object

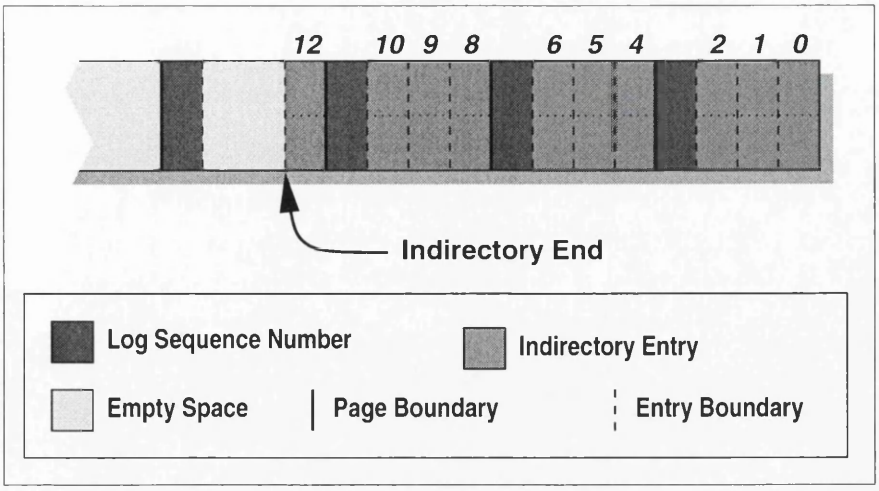


Figure 4.5: Layout of Indirectories

is allocated, the indirectory will have to grow (entry **5** will be created). Entries **1**, **2**, and **4** point to objects **c**, **a**, and **b** respectively. However, since object **a** has a reference count of 0, in the next garbage collection it will be collected, provided there are no references to it from inside this partition.

Figure 4.5 illustrates the layout of the indirectory pages (for clarity reasons, the figure assumes that only 4 indirectory entries can reside on each page). Each such page contains an LSN that is used by the recovery manager (⇝§4.4.3 and §4.4.3.2). The LSN is placed at the beginning of the page, taking the place of an indirectory entry (the size of both entities is currently the same: 8 bytes). For simplicity reasons, no attempt is made to re-use the indirectory indexes taken up by the log sequence numbers. The effect that this has on the the maximum number of indirectory entries, and hence the maximum number of objects, inside each partition is minimal (currently only 32 entries out of 32K are "wasted" because of this).

4.5.4 Object Layout

Figure 4.6 illustrates the layout of objects in Sphere. Each object has an 8-byte header that contains the following three fields.

- **Size (26 bits)** — The object size in 8-byte chunks. This forces the object size to always be 8-byte aligned. If more accuracy is required, it is up to the kind implementation to include the real size somewhere in the object's contents. This limits the maximum Sphere object size to $2^{26+3} = 128\text{MB}$.
- **Kind (6 bits)** — The kind of the object. This limits the maximum number of co-existing kinds in the same store to $2^6 = 64$.
- **Info Field (32 bits)** — The use of this field is kind-specific. Typically, this is where any information related to the type of the object is stored (notice that objects of the same kind can have different types, i.e. scalar arrays might be arrays of integers or characters).

Another issue, illustrated in figure 4.6, is the fact that the offset field on the indirectory entry points to the beginning of the object header, rather than the beginning of the object contents.

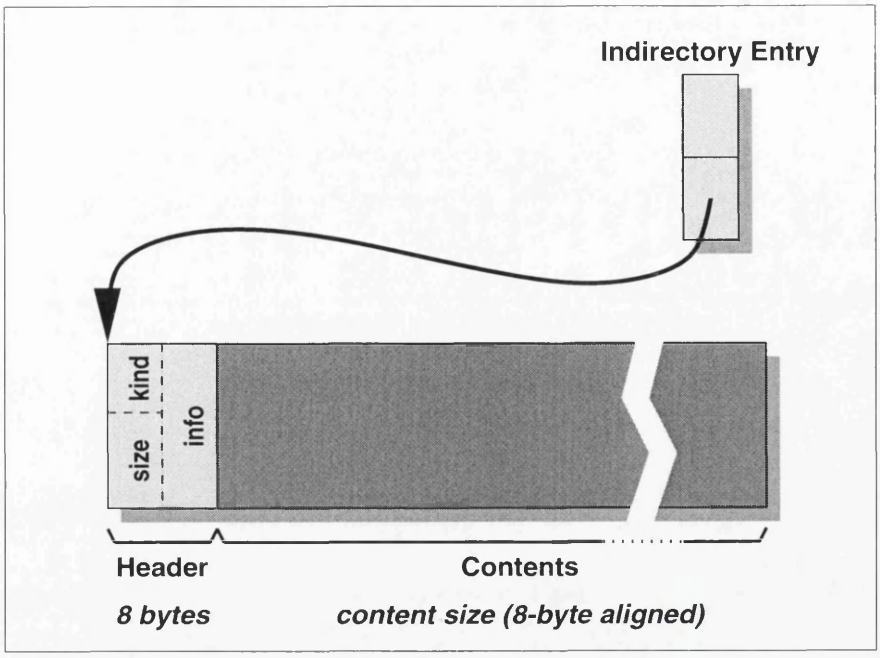


Figure 4.6: Layout of Objects

4.6 Basic Block Management

This section describes the mechanism employed to abstract the presence of multiple store segments (defined in section 3.4.4) from the rest of Sphere. Section 4.6.1 presents the implementation of the segment table, employed to manage segments and provide them with unique identifiers. Section 4.6.2 presents the scheme employed to uniquely address BBs inside each segment. Finally, section 4.6.3 outlines the BB allocation/deallocation mechanism.

4.6.1 The Segment Table

The *Segment Table* (ST) is an in-memory table that contains one entry per used segment. For simplicity, its maximum length is currently fixed to 32 entries, but this can be trivially extended. Each entry, if allocated, includes the name of the device (i.e. file or a raw partition) that contains the corresponding segment, the operating system device descriptor, and the segment's maximum size (number of BBs included in the segment). The index of its corresponding entry in the ST gives each segment a unique identifier, called the *Segment Identifier* (SID). The operation of the ST is illustrated in figure 4.7.

When a Sphere store is created, the ST is constructed from the segment information included in the Sphere configuration file (⇝§C) and then the store segments are created and initialised. When an already-existing store is opened, the ST is again constructed from the contents of the configuration file. However, before normal operation can commence, Sphere has to verify that *(i)* no segment is missing (as Sphere cannot operate otherwise) and *(ii)* the segments enumerated in the configuration file are indeed the correct segments for that particular store. For this, the header of the bootstrap segment contains a table specifying which store segments should be present for that store. Additionally, the header of each segment contains the following information.

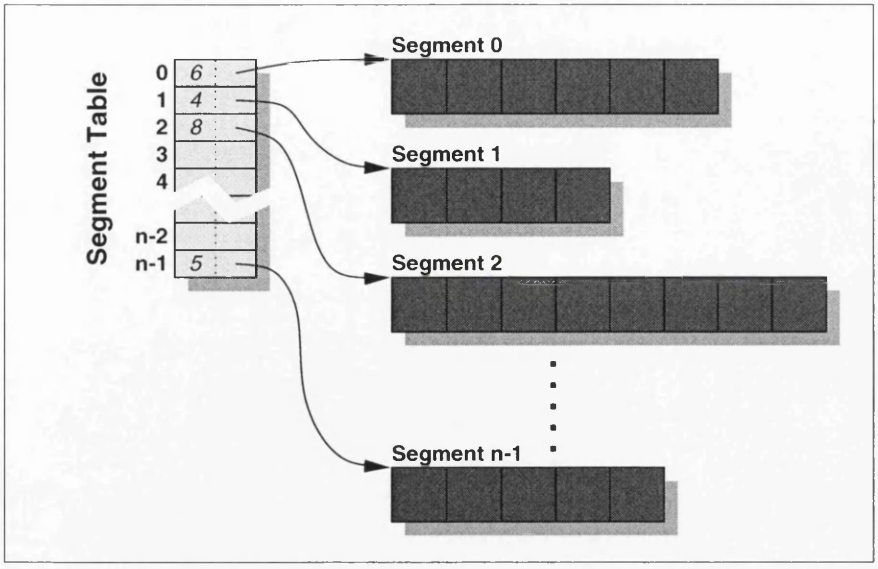


Figure 4.7: The Segment Table

- A *magic number* that ensures, with a low probability of error, that the segment is indeed a Sphere segment. The magic number chosen for Sphere segments is `0xDEADD0D0`.
- The SID, which has to match the one given in the configuration file for that segment.
- A *fingerprint*, calculated when the store is created. All segments of the same store should have matching fingerprints on their headers.

  This fingerprint is constructed from information such as user name, host name, date of creation, time of creation, etc. The MD5 algorithm is used for this, which guarantees that, if two fingerprints have been constructed from different information, there is an extremely low probability of them being the same.

Notice that the name of the device holding the segment is not recorded on the segment header, so that it can be moved easily, if necessary.

Given all of the above, Sphere can perform all the necessary checks to ensure that all the store segments are present. It also supports the enlargement of existing segments and the addition of new segments. The store initialisation procedure is outlined below.

❶ Initialise the ST from the contents of the configuration file.

❷ Open the bootstrap segment (its SID is 0), which should always be present, and validate it by performing the checks described in step ❹.

❸ Ensure that the SIDs of the segments present in the ST match the information contained in the bootstrap segment header. If there are extra segments in the ST than what is recorded in the bootstrap segment header, then these are added, extending the store; this is done in step ❺, after all initialisation has successfully finished. On the other hand, if there are segments missing from the ST compared to what is recorded on the bootstrap segment header, initialisation fails.

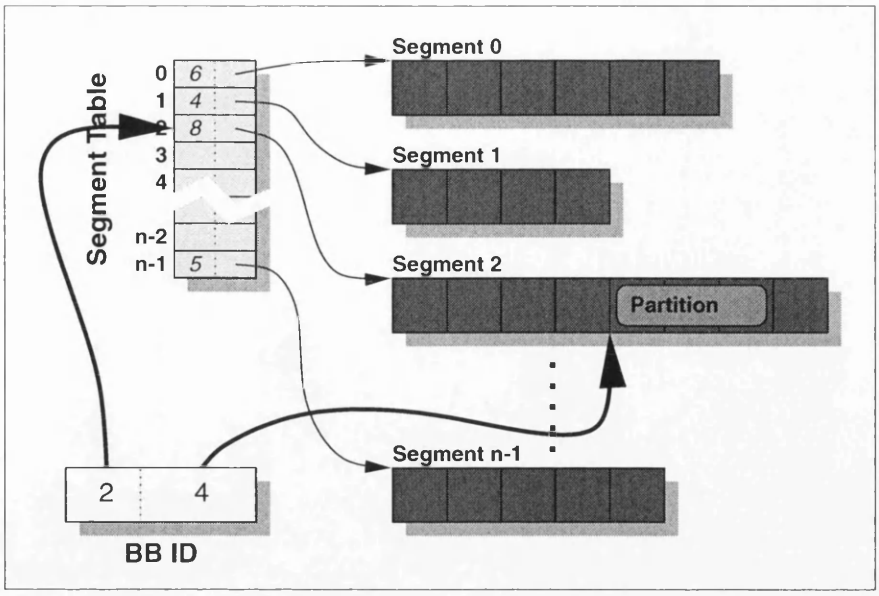


Figure 4.8: Basic Block Addressing

❹ Open each other segment present in the ST and validate it by ensuring that the magic number on its header is correct and the SID on its header matches the one of its corresponding ST entry. If the maximum segment size in the ST is greater than the one recorded on the segment header, then its maximum size is updated appropriately. On the other hand, if it is smaller, initialisation fails.

❺ Create all the new segments, if any.

When the above procedure has completed successfully, it is safe for Sphere to commence normal operation. It is worth pointing out that it could have been possible to include all the segment information in the bootstrap segment. This approach could possibly have reduced the amount of necessary consistency checks, as the store could be trusted to contain correct information, unlike the user who might erroneously edit the configuration file. However, this would have required the user to apply any updates to this information (extending segment maximum size, adding new segments, and moving a segment to a different device) by using a specialised utility. It was decided that it would be simpler to achieve this by changing the configuration file instead; this can be achieved by using any standard text editor.

4.6.2 Basic Block Addressing

As BBs need to be accessed by several Sphere modules, it was necessary to adopt an addressing mechanism for them. For maintainability reasons, such addressing scheme had to "hide" the presence of multiple segments from most other Sphere components. It was also necessary to fit each *BB Identifier* (BBID) in a single 32-bit word, in order to keep the PT entry length small. Because of this, it was not possible to use a combination of the SID and the offset of the BB inside the segment, since the resulting BBID would have been too wide. Instead a combination of the SID and the *index* of the BB inside the segment is used.

Assuming that the BB size is 512KB and the maximum segment size is 2GB, the maximum number of BBs inside a segment is $2\text{GB}/512\text{KB} = 4,096 = 2^{12}$. Therefore, it is possible to allocate the 20 out of the 32 bits of a 32-bit BBID for the BB index, leaving enough room to be able to deal with a future increase in

the number of BBs per segment, and used the remaining 12 bits to house the SID (these will be more than adequate for this purpose, as currently the maximum number of segments is $32 = 2^5$).

Figure 4.8 illustrates the BB addressing mechanism. The BB illustrated, which happens to correspond to the beginning of a partition, is the one with index 4 (indexes start from 0) in the segment with SID 2.

4.6.3 Basic Block Allocation

When a partition-allocation request needs to be satisfied, a number of free BBs must be located in the store. These BBs should be contiguous and reside in the same store segment. In this way the pages of the partition are located close to each other[6] in an attempt to increase the efficiency of large accesses to the same partition (i.e. fetching of well-clustered objects, garbage collection, etc.).

As described in section 3.8.2, the free-space management of segments relies on a bitmap, with each of its bits representing one BB. If that bit is 0, the corresponding BB is free, otherwise it is allocated. Each segment has its own bitmap, which resides entirely on a single page. This page is always kept pinned in memory (↝§4.7) to prevent it from being evicted during execution and causing unnecessary I/O traffic.

When a new partition needs to be allocated, its size is first determined by a regime-specific operation. Then, the bitmap for the first segment is scanned for a contiguous number of 0s that correspond to the calculated size. If they are found, they are set to 1s, the appropriate log record is written to the log, and the BBID of the first BB, which corresponds to the beginning of the partition, is returned. If they are not found, the bitmap of the next segment is scanned, until either the allocation succeeds for some segment or fails for all segments. In this case, the store is considered full. Currently, Sphere first attempts to allocate a new partition in the last segment where an allocation succeeded and, if this fails, cycles through the rest of the segments in the order they appear in the ST. The allocation policy on the bitmap is a simple first-fit [WJNB95].

When a partition needs to be de-allocated, the corresponding bits on the bitmap of its segment are set to 0s and the appropriate log record is written to the log.

4.7 The Disk Cache

The *Disk Cache* (DC) is responsible for caching disk pages and writing them back to disk, if they are dirtied. The address of a page is a combination of a BBID and an offset. The latter is allowed to extend into subsequent BBs. There are two kinds of data that need to be addressed in Sphere: *(i)* data inside a partition and *(ii)* bootstrap data on the segment header. The above addressing scheme satisfies both of these since *(i)* all partition accesses are relative to the first BB of the partition (this is called the partition location) and *(ii)* all segment header accesses are relative to the first BB of the segment.

The above addressing scheme however raises the problem that it does not provide each page with a unique address. Assuming that the BB size is 512K, consider the following two BBID/offset combinations: 2/10 and 1/(512K + 10). Clearly, even though they are different, they refer to the same page. To avoid fetching the same page twice, it was necessary to use a unique address for each page. So the DC translates internally the

[6]This is actually open to debate, since modern disk controllers cannot guarantee that, what seem to be adjacent disk pages to the operating system, are actually adjacent pages on the disk surface. However, there is experimental evidence that indicates that linear reads over a disk are considerably faster than random ones (↝§5.3.5).

address of a page from a BBID/offset combination to the corresponding SID and offset inside the segment. In the above example, the DC will translate both addresses to SID 0 and offset (1024K + 10).

It is obvious that the DC is aware of the presence of segments. In fact, it is the only component, apart from the ST, that accesses them directly. Every other component uses BBIDs and offsets, without being aware which I/O device will be accessed.

In the following three sections three different aspects of the DC implementation are presented: the use of page descriptors (↝§4.7.1), the dirtying operation of the contents of a page (↝§4.7.2), and the MT-safety aspects of the DC (↝§4.7.3). Finally, some measurements are presented in section 4.7.4.

4.7.1 Page Descriptors

Each cached page has a corresponding *Page Descriptor* (PD) (not to be confused by the descriptors, defined in section 3.5.3). This is a 60-byte data structure containing status information about the page. Among other things, it includes the following.

❶ Page address (SID and offset into the segment).

❷ Pointer to a buffer containing the page data.

❸ Handle to the device containing the segment.

❹ Next pointer for linking into the hash table (see below).

❺ Dirtying information: dirty flag, recovery LSN, and LSN offset inside the page (↝§4.7.2).

❻ Pinning information: pinning count and latch (see below).

❼ Contents status flag: fetching/ready (↝§4.7.3).

❽ Usage flag for the clock algorithm (see below).

When a Sphere module dispatches a page request, the DC returns a pointer to the appropriate PD. Through the PD, the module can access the contents of the page. It can also update them, if necessary, notifying the DC that it has done so, by updating the dirtying information on the DC appropriately (↝§4.7.2). Every time a page is requested, the pinning count on its PD is increased by one. When the module does not need the page anymore it notifies the DC, which decreases the pinning count. No page is evicted while its pinning count is greater than 0, therefore this ensures that no page is evicted while being used by a Sphere module. All updates to the pinning count are performed using the CAS operation (↝§4.7.3).

It was necessary to optimise the look-up operation for the PD of a page, as a linear search over all PDs would have a negative performance impact. For this reason, a hash table (HT) was added. The hash value of a PD is a combination of the page address (SID and offset), truncated to the size of the HT. All the PDs that are hashed on the same entry are linked, with the head of this list residing on their HT entry.

The eviction mechanism adopted in the DC is the widely-used *Clock Algorithm* that emulates the least-recently-used policy [Tan87, GR93]. For this purpose, two pointers pointing to different PDs were introduced in the DC that correspond to the primary and second "hands" of the clock algorithm. Access to them is guarded by a mutex (the hands lock, ↝§4.7.3). Additionally, each PD contains a usage flag that is set

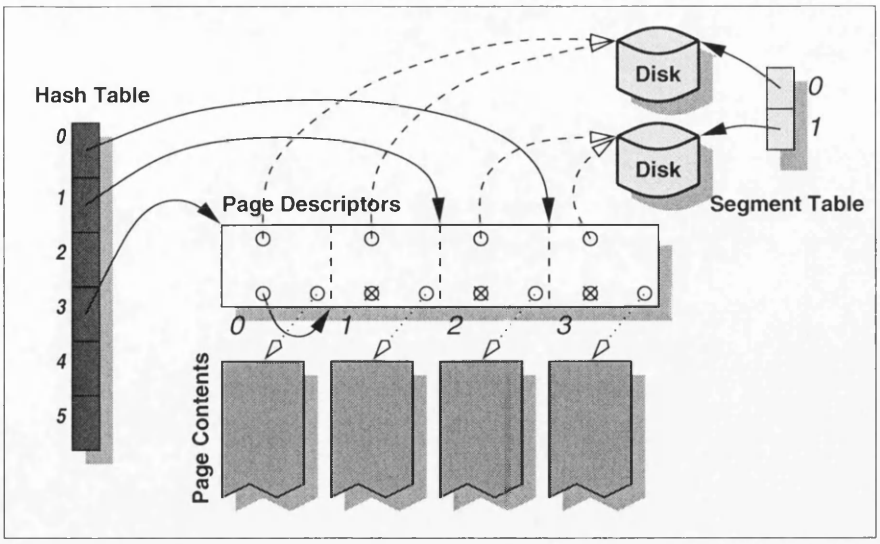


Figure 4.9: Organisation of the Disk Cache

when a request for the corresponding page is raised and cleared as the secondary "hand" moves forward during eviction.

The structure of the DC is illustrated in figure 4.9. For simplicity reasons, the figure assumes that the store comprises two segments, 0 and 1, the DC can only accommodate four pages, and the length of the HT is six. In the figure, PDs 0 and 1 have been hashed on the same hash table entry, namely 3, with the next field on PD 0 pointing to PD 1, while PDs 2 and 3 have been hashed on entries 1 and 0 respectively. Additionally, each PD points to its corresponding I/O device (PDs 0 and 1 correspond to pages residing on segment 0 and PDs 2 and 3 to pages on segment 1).

4.7.2 Dirtying Pages

The procedure of dirtying a page is outlined belowas follows.

❶ Request a page and get the corresponding DC. At this point the page has been pinned and cannot be evicted by another thread.

❷ Send a log record request to the log manager (⇝§4.4.2) and accept the corresponding LSN.

❸ Apply the necessary update on the page contents.

❹ Store the recovery LSN in the PD and set the dirty flag.

❺ Release (unpin) the page. Only after this point the page can be evicted and its contents written to disk.

When the page is eventually written to disk, either by being evicted or during shutdown, the log has to be forced up to the location corresponding to the recovery LSN on the PD to ensure the correctness of the recovery operation (⇝§4.4.3). Additionally, there are some pages on which logical updates take place (e.g. indirectory pages). It is necessary that the LSN corresponding to the last update applied on them is stored on them before they are written to disk (⇝§4.4.3.2). For this purpose, the offset of the LSN location on the page is stored in the corresponding PD so that eviction-time the DC stores the LSN automatically on the page.

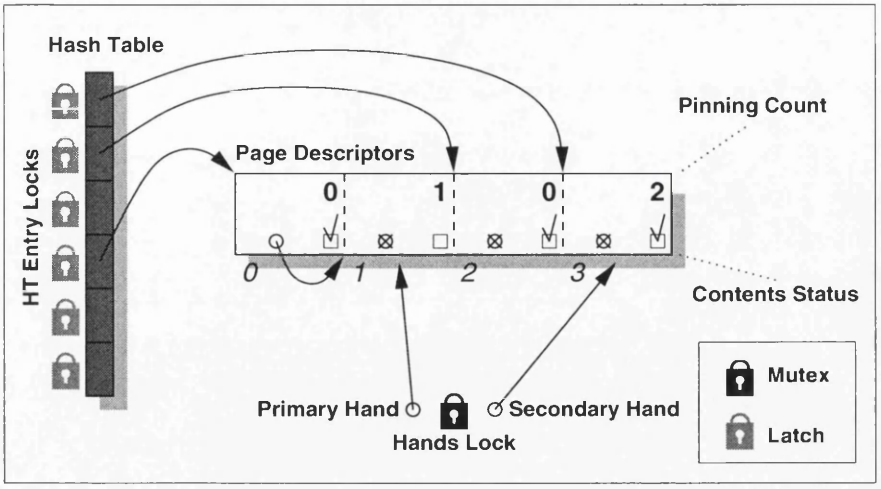


Figure 4.10: Locks of the Disk Cache

4.7.3 Multi-Threaded Safety

Figure 4.10 illustrates all of the locks that have been included in the DC data structures. They are the following.

- **Hash Table Entry Locks (Latches)** — A thread takes the HT entry lock when iterating over the corresponding HT chain. Additionally, while such a lock is being held, no other thread is allowed to remove an entry from the corresponding HT chain. This ensures that, when a chain is iterated over after the lock has been taken, the chain is always in a consistent state.
- **Hands Lock (Mutex)** — It is taken during eviction, before the "hands" of the clock algorithm are manipulated, and is retained until an evictable page has been found. A side-effect of this is that, while holding the hands lock, no pages are evicted by other threads, hence no PDs are removed from HT chains.

Additionally, each PD contains a pinning count, denoting how many threads are using the corresponding page at any time (this is updated atomically using CAS, ⇝§4.2), and a contents status flag denoting whether the contents of the page are installed properly or are being fetched.

After a page request is raised, the procedure to discover its PD or fetch the page from disk is outlined below. In the description, *new page* is the page that is being looked up and *old page* is the one that will be evicted in order to make space for the new one.

❶ Take the HT entry lock for the new page and iterate over the HT chain looking for the PD. If found (this is a **Cache Hit**), increase the pinning count on the PD using CAS, release the HT entry lock, set the usage flag on the PD, and return the PD.

❷ If not found, release the HT entry lock and take the hands lock. During the time the hands lock is held, other threads can discover pages already cached in the DC, as the operation of step ❶ does not require the hands lock. However, no other thread can evict a page.

❸ While holding the hands lock, iterate again over the HT chain for the new page looking for the PD, just in case another thread inserted it, after this thread released the HT entry lock and before it took

the hands lock. Since the hands lock is being held, no other thread can remove an entry from the chain, so it is safe to iterate over it. If the PD is found (this is a **Secondary Hit**), increase the pinning count on it using CAS, release the hands lock, set the usage flag on the PD, and return the PD.

❹ At this stage, the page is definitely not in the cache (this is a **Cache Miss**), therefore it has to be fetched. As the hands lock is still being held, the clock "hands" can be manipulated. Using the clock algorithm, find an available page, i.e. a page that is not pinned and its usage flag is not set. Before checking the pinning count of the page, its corresponding HT entry lock has to be taken, to avoid another thread pinning it concurrently. Once an available entry is found (the number of entries checked before one is found are called the **Searching Steps**), it is unlinked from the HT chain, while still holding its HT entry lock. If the page being evicted is dirty, the HT entry lock is retained (see step ❼), otherwise released. If at this point, another thread is looking for the page that has just been evicted, it will not find it, will attempt to take the hands lock in order to fetch it, and eventually will find it during step ❸.

❺ While still holding the hands lock, copy the new SID and segment offset (i.e. the address of the new page) on the PD of the newly-evicted old page, set its contents status flag to "being fetched", increase its pinning count (guaranteed to be exactly one at this stage, since no other thread can access the PD as it is not chained yet), and insert it in the chain corresponding to the new page. This does not require the HT entry lock, as it can be done atomically (by inserting it at the beginning of the chain after having set up its next field). Again, if another thread is looking for this page, it will not find it but will discover it during step ❸, after eventually taking the hands lock.

❻ At this point the hands lock is released, so other threads can evict pages and they are not block during the I/O operations that this thread will do.

❼ If the evicted page is dirty, its HT chain lock should have been retained (see step ❹). In this case, the PD should contain the new SID/offset and should have been inserted into the HT chain of the new page (see step ❺), but the page contents should still contain those of the old page. While holding the old HT chain lock, force the log and write the old contents to disk. Holding the HT chain lock prevents other threads from fetching again the page that is being evicted, while the disk writes are taking place, so that the page contents are not read inconsistently. Once these I/O operations have completed, the old HT chain lock is released.

❽ The contents of the new page can now be read from the disk. Once the I/O operation is finished, the contents status flag is set to "ready". If other threads have found the PD, after it was inserted in the HT chain and before the I/O operation was finished (this can only happen at steps ❶ and ❸), they would block while the the contents status flag is set to "being fetched" (this is called **Waited For Contents**), but after having released any locks that they had taken, in order not to prevent other threads from operating.

The above algorithm, even though complex, has the important property that the order in which locks are taken prevents deadlocks. A deadlock can occur when two threads take the same two locks in the opposite order. This cannot take place in the above algorithm, as the only time when a thread takes a second lock (step ❹, when the HT entry lock of the page that is being evicted is taken), it is always the case that the hands lock had been taken first.

	SingleBench		MultiBench	
	Number	% of Total	Number	% of Total
Cache hits	2,007,149	99.8578	7,248,210	98.4687
Secondary hits	0	0.0000	995	0.0135
Cache misses	2,859	0.1422	111,723	1.5178
Total	2,010,008		7,360,928	
Waited for contents	0	0.0000	910,693	12.3720
Allocations*	2,859		111,723	
Total searching steps	3,718		111,812	
Average searching steps	1.30		1.00	

* This is the number of times a PDE has been allocated and it is the same to the number of misses. It is replicated for clarity reasons.

Table 4.1: Measurements of DC Usage

4.7.4 Measurements

Table 4.1 presents the DC measurements taken from the two benchmarks introduced in section 4.3. In both cases the percentage of DC hits is very high (99.6% and 98.5%). This was aided by the synthetic nature of the benchmark and the layout of objects on disk. However, it indicates that all threads are operating over the lists roughly at the same speed, hence they are visiting the same pages. This is encouraging as it suggests that no threads starved or were delayed by the concurrency control. Only MultiBench had secondary hits or calls that waited for contents (note that this is quite high at 12.4%). However, this was to be expected as both these events only happen in a multi-threaded environment.

4.8 The Partition Table

The partition table provides one level of indirection to the physical location of partitions, as described in section 3.6.1. Additionally, as described in the following sections, it is also a convenient place to add some extra functionality. Section 4.8.1 describes the format of the PT entries and section 4.8.2 illustrates the organisation of the PT on disk. Section 4.8.3 outlines the management of free entries on the PT and sections 4.8.4 and 4.8.5 describe respectively the mechanisms for discovering partitions open for allocation and locking partitions for updates, promotion, and garbage collection.

4.8.1 Partition Table Entries

Each *Partition Table Entry* (PTE) is 16 bytes wide and is made up of four 4-byte fields: *(i)* the *Partition Location* field, *(ii)* the *Flags* field, *(iii)* the *Update Information* field, and *(iv)* the *Garbage Collector Information* field. Figure 4.11 illustrates the contents of these fields, which are also described below.

- **Partition Location Field** — The BBID corresponding to the first BB of the partition. The full width of this field (32 bits) is taken up by the BBID.
- **Flags Field** — It contains information on the status of the partition. Its layout is illustrated in figure 4.11.b. The following five fields are included.

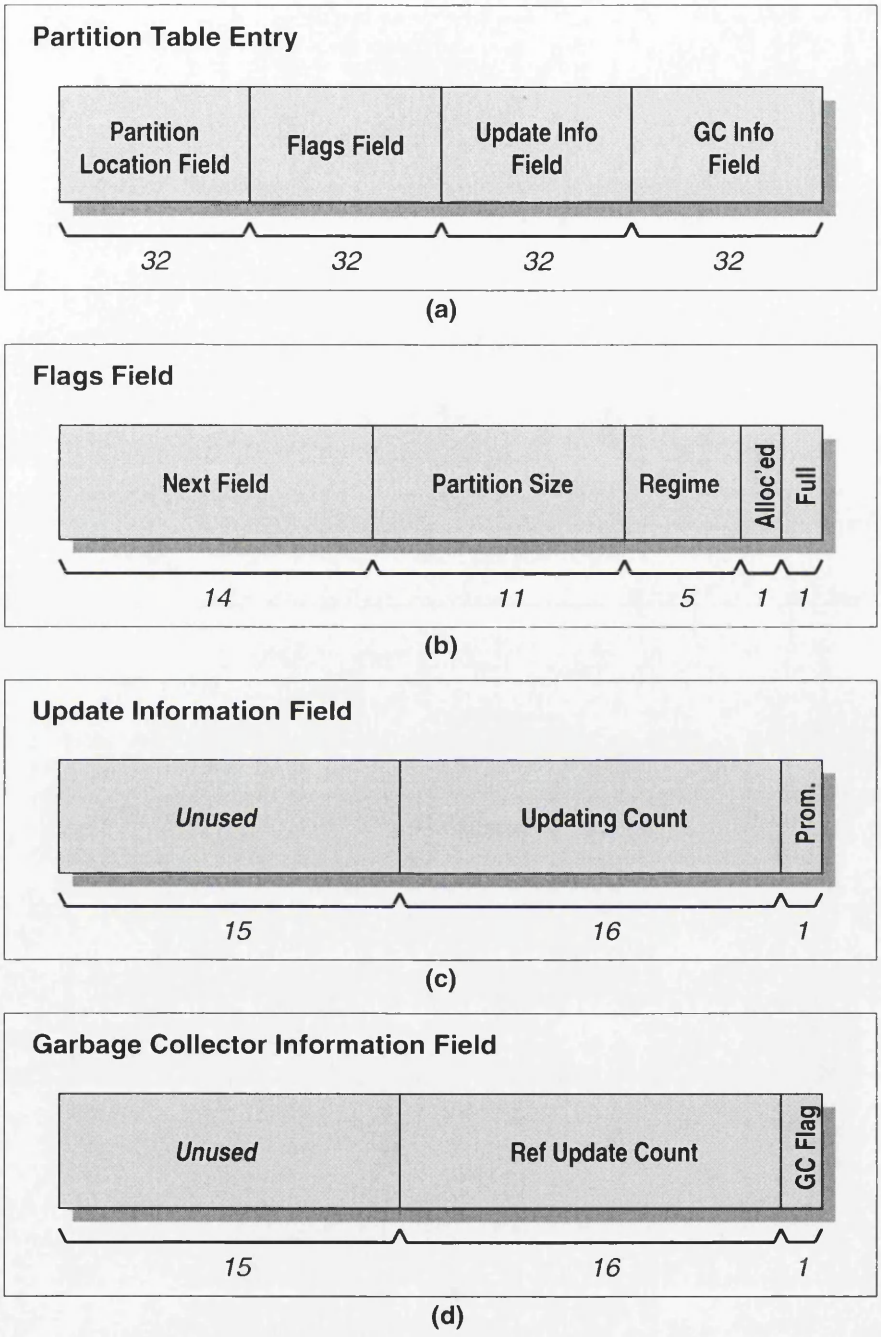


Figure 4.11: Partition Table Entry

- **Next Field (14 bits)** — it is used when the PTE is linked in one of the two lists described in sections 4.8.3 and 4.8.4. Given that the Sphere page size is 8K and that this next field needs to point to other entries inside the same page, 14 bits can accommodate an offset inside an 8K space (13 bits) plus a null value. If the page size increases, it will be possible to use indexes, rather than offsets, and still be able to accommodate the next field in 14 bits.
- **Partition Size (11 bits)** — the size of the partition in number of BBs. Given that the current BB size is 512K, this assumes that the maximum partition size is $2^{11} \times 512\text{MB} = 1\text{GB}$. This is more than adequate for the current needs of Sphere.
- **Regime (5 bits)** — the managing regime of the partition. This limits the number of regimes that can co-exist in the same Sphere store to $2^5 = 32$.
- **Allocated Flag (1 bit)** — it denotes whether this PTE has been allocated or not.
- **Full Flag (1 bit)** — it denotes whether the partition is full or can still satisfy allocation requests.

- **Update Information Field** — It contains information on whether the partition is being updated and objects are being promoted to it. Its layout is illustrated in figure 4.11.c. It includes the following three fields.
  - **Updating Count (16 bits)** — it denotes the number of threads that are updating the partition. It assumes that less than 64K threads are updating objects on the same partition.
  - **Promotion Flag (1 bit)** — it denotes whether objects are being promoted to the partition.
  - **Unused (15 bits)** — currently unused.

- **Garbage Collector Information Field** — It contains information needed by the garbage collector. Its layout is illustrated in figure 4.11.d. It includes the following three fields.
  - **Reference Update Count (16 bits)** — it denotes how many references pointing to that partition have been updated since the last garbage collection of that partition. It is used to determine which partition to garbage collect next, according to the Cook *et al.* scheme [CWZ94].
  - **Garbage Collection Flag (1 bit)** — it denotes whether the partition is being garbage collected.
  - **Unused (15 bits)** — currently unused.

The information described above is placed in the PTE, rather than in the partition header, in order to avoid having to fetch another page in order to accesses it. For example, in order to access an indirectory entry, the partition location needs to be known, as well as the size of the partition (since the indirectory is at the end). Placing the partition location and size in the PTE allows them to be accessed with at most one page-fault.

Given the above layout of the fields, it is obvious that the information in a PTE could in fact fit, with a few modifications, into three 32-bit words, rather than four. However, it was decided to use four so that there is an integral number of PTEs per PT page (↝§4.8.2) and also to provide the two unused fields to accommodate any future expansion.

4.8.2 Layout on Disk

The maximum number of partitions inside the store is determined by the number of bits in the PIDs that are used to store the PI (↝§3.7.1). This is currently set to 16, therefore the maximum number of partitions in a

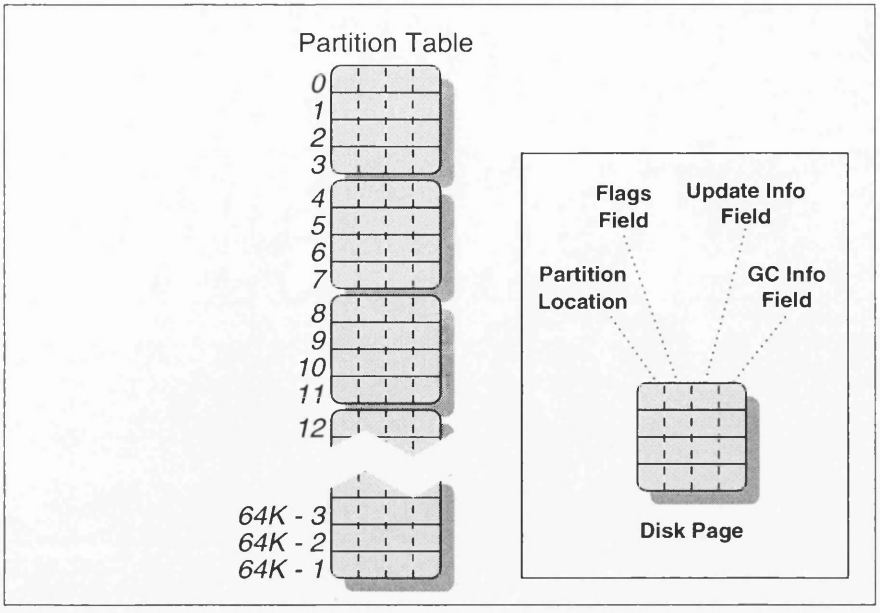


Figure 4.12: Layout of the Partition Table

Sphere store is $2^{16} = 64$K. As each partition entry is 16 bytes, the size of the PT is 16×64KB $= 1$MB.

It would have been inefficient to keep the entire PT constantly cached into memory, since it is likely that only a small proportion of the partitions will be active at any time. Therefore, it was decided that its pages will be faulted-in lazily. Since the Sphere page size is currently 8K, the PT spans over 128 pages and each page contains 512 PTEs. The layout of the PT on disk is illustrated in figure 4.12 (for simplicity reasons this figure and the ones in the following two sections assume that only 4 PTEs fit on each PT page).

4.8.3 Managing Free Entries

When a new partition needs to be allocated, a free PTE must be discovered. A sequential search over the entire table would be too inefficient, since several page-faults might be caused, as several PT pages could be visited before an appropriate PTE is found. Instead, a more efficient scheme was adopted.

All free PTEs are linked into free-lists and the next field on each PTE is used to store the next PTE on the list. There is one such list for each page of the PT. The heads of the lists reside on a single page, called the *Partition Table Directory* (PTD) that is kept constantly pinned in memory. When a free PTE needs to be discovered, the heads of the free-lists are scanned sequentially, until one with a nun-null value is found. This guarantees that the corresponding page contains at least one free PTE and only that page will need to be fetched from the disk (if it is not already resident). The PI of the newly allocated partition is the index of its PTE.

Figure 4.13 illustrates the use of the PT free-lists. Notice that all of the free PTEs appear on one of the lists and that the partition space itself does not need to be touched when PTEs are added or removed from the free-lists.

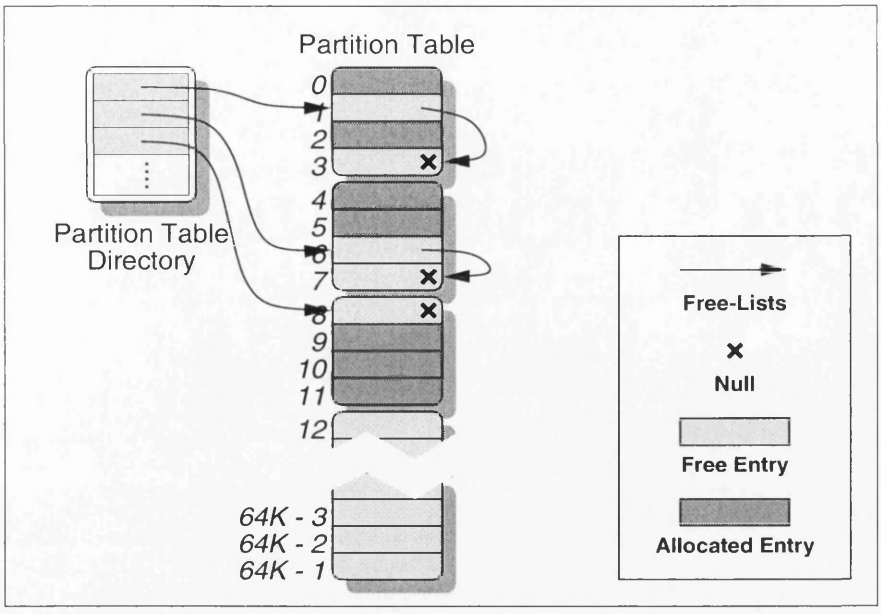


Figure 4.13: Managing Free Partition Table Entries

4.8.4 Discovering Partitions Open for Allocation

When an object allocation request is raised, it is necessary to retrieve a partition that is open for allocation and that matches the regime where the object should reside. Again, it would be too inefficient to linearly scan the entire PT to find an appropriate partition. Instead, the partitions that are open for allocation are linked into linked lists, one list per regime and page of the PT. The heads of these lists also reside on the PTD page. It is worth pointing out that there might be more than one partition open for allocation for a given regime (e.g. the garbage collector might empty several partitions, before more objects are allocated).

Upon an allocation request, the regime where the object should be allocated is specified. Then, the heads of the lists corresponding to that regime are scanned linearly, until one with a non-null value is discovered. If no appropriate partition is found, a new one that implements the required regime is created.

Figure 4.14 illustrates the use of the allocating lists. For simplicity, it is assumed that only 2 regimes co-exist in the store. On the PTD there are two allocating list heads (one per regime) for each PT page, as well as one head for the free-list (described above). Again, the next field on the PTEs is used for linking purposes. Notice that not all allocated PTEs appear on an allocating list; only the ones that are open for allocation do (e.g. 2, 5, 9, and 10) and not not the full ones (0, 4, and 11).

However, it might be more appropriate to entirely fill a partition first, before start allocating on another one[7]. Using only the allocating lists, as described above, cannot guarantee this, as between allocations the garbage collector might empty one partition entirely and its PTE might be inserted into the allocating list, before the one that has been actively allocating. To deal with this, a new table has been introduced. It is called the *Actively Allocating Partition Table* (AAPT) and contains one entry per regime. Each APPT entry points to the partition that is actively allocating for that regime.

[7]This does not take into account clustering considerations.

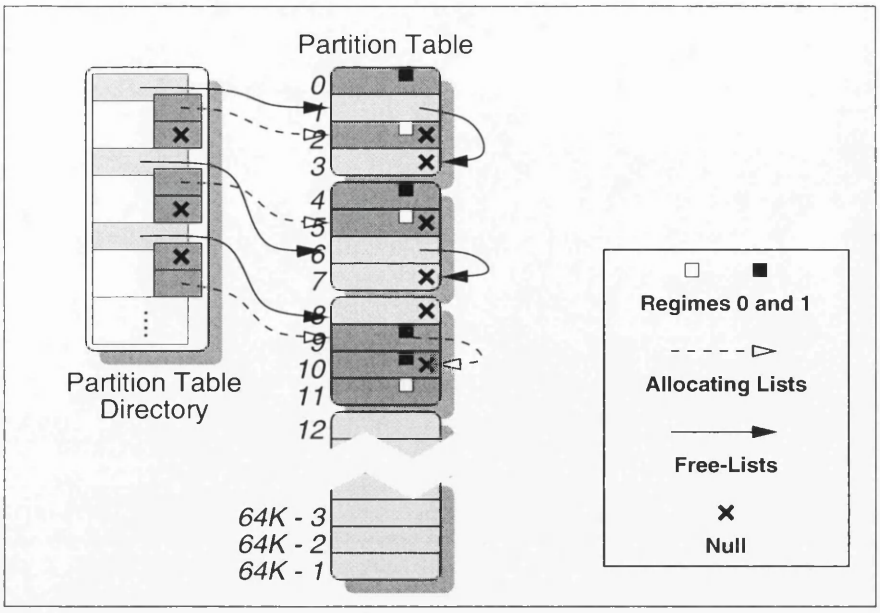


Figure 4.14: Discovering Partitions Open for Allocation

When an object allocation request is raised, the AAPT is looked up first. When the partition referenced by this table becomes full, a new partition open for allocation is retrieved and the entry of the AAPT is updated to point to it. It is worth pointing out however that if multiple threads need to concurrently allocate objects to partitions of the same regime, each thread will have to use a different partition (⇝§5.2.2) and will update the AAPT with the next available partition of that regime (or create a new one if there are no allocated ones available).

The AAPT resides on the boot segment, on a page that is constantly pinned in memory. Its operation is illustrated in figure 4.15 (partitions 5 and 9 are actively allocating for regimes 0 and 1 respectively).

4.8.5 Locking Partitions for Updates

To ensure MT-safety, all updates to the PT are guarded by a single lock (mutex). As they are expected to be relatively infrequent, this should not cause any concurrency problems. However, if this is proved not to be the case, a more sophisticated scheme can be easily adopted (e.g. a lock per PT page).

There are four update types that can be applied to a partition.

- ❑ **Object content updates**
- ❑ **Reference count updates**
- ❑ **Object allocation (promotion)**
- ❑ **Garbage collection**

The invariants that must be obeyed, when partitions are updated, are the following.

❶ When garbage collection is taking place on a partition, no other updates on that partition are allowed.

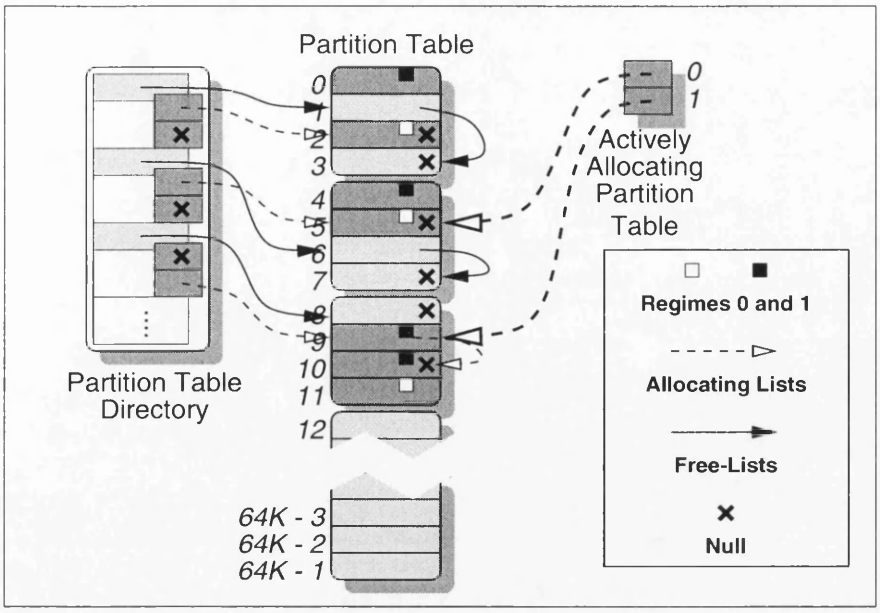


Figure 4.15: Actively Allocating Partition Table

❷ Only one thread at any given time can be promoting objects to a partition.

❸ Only one thread at any given time can be updating reference counts on a partition (to ensure that their values are updated correctly).

❹ Multiple threads are allowed to be updating concurrently object contents on the same partition. This assumes that the mutator provides the appropriate concurrency control so that the all object contents are updated correctly (⇝§3.1).

❺ Threads that update object contents are allowed to operate over a partition along with a possibly different one performing a promotion and another different one performing reference count updates, as long as the items ❷, ❸, and ❹ are obeyed.

The above are enforced using the following flags and locks.

❑ **Updating Count (Update Information Field)** — It denotes the number of threads that are updating the partition; this includes object content and reference count updates. A thread has to increase this count before it starts performing such updates. If the count is 0, it can only be increased to 1 if the garbage collection flag (see below) is not set. If the count is not 0, it can always be increased.

A thread that increases the updating count can only decrease it after its history has been committed. This ensures that, when the count is 0, all updates on that partition have been committed, therefore a garbage collection can be initiated if necessary (⇝§6.3.1.1).

❑ **Promotion Flag (Update Information Field)** — It denotes that a thread is allocating new objects to the partition. A thread has to set this flag before it starts allocating objects and can reset it only after it has committed the corresponding history. The promotion flag can only be set by one thread at a time and cannot be set while the garbage collection flag is set.

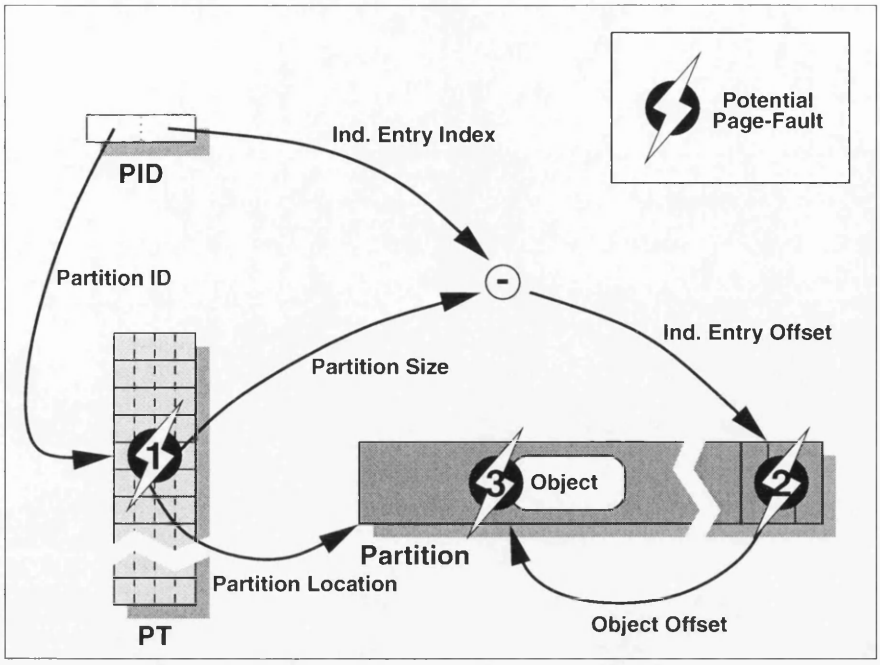


Figure 4.16: Full Object-Fault

- **Garbage Collection Flag (Garbage Collection Information Field)** — It denotes that the partition is being garbage collected. Only the garbage collector thread can set this (⇝§6.3.1.1) and it has to do so before it starts a new garbage collection cycle on the partition. This flag cannot be taken while the updating count is greater than 0 or while the promotion flag is set.

- **Reference Count Lock** — This resides on the partition location cache entry for the partition (⇝§4.9.2). It has to be taken before any reference counts are updated on that partition (see section 4.10 on the mechanism that performs such updates).

4.9 The Location Caches

Because of the layout of partitions in Sphere (⇝§4.5.2), a single object-fault might cause up to three page-faults. Furthermore, according to the Sphere API, for an object to be fetched, the mutator has to interact with Sphere twice: *(i)* the `sphere_inspectObject` call (⇝§A.2.7) first returns the kind and regime of the object and its size, so that enough memory can be allocated by the mutator, and *(ii)* the `fetch` operation (⇝§A.3.3) on objects (dispatched based on the kind and regime returned from the previous call) performs the actual object-fault. The former has to access the object header (⇝§4.5.4), the latter the object contents.

In an attempt to optimise the object-faulting operation, some caching mechanisms were introduced in Sphere. They are described in the following sections. Section 4.9.1 illustrates the operation of a full object-fault and section 4.9.2 introduces a cache for partition location. Finally, section 4.9.3 describes a cache for object location that was later removed as it was shown not to be beneficial.

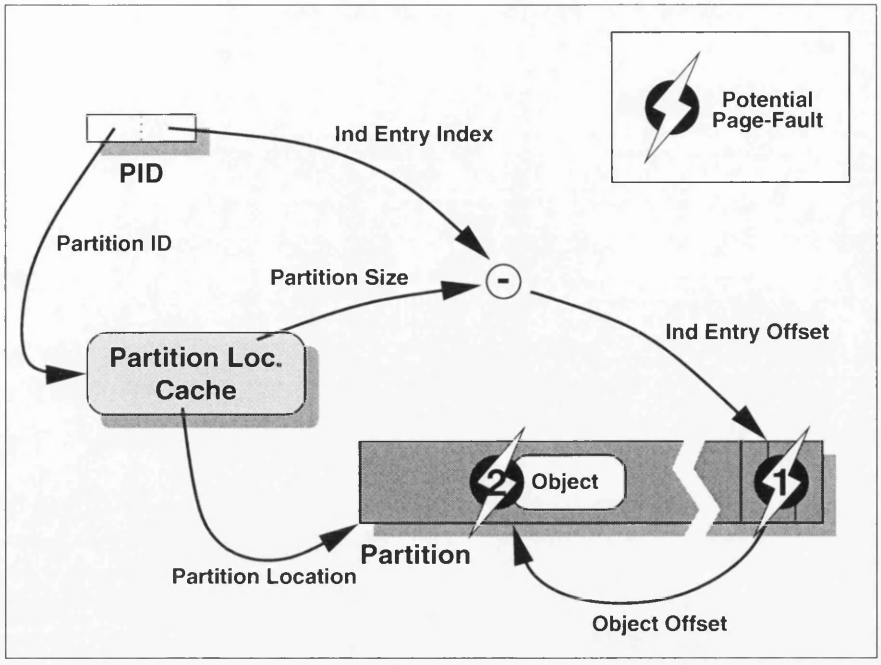


Figure 4.17: The Partition Location Cache

4.9.1 Full Object-Fault

Figure 4.16 illustrates the mechanism of accessing the header and contents of an object. Given the object's PID, the PI is extracted and its corresponding PTE retrieved — this might cause a page-fault. The PTE contains the partition size and location. The indirectory index is extracted from the PID and, in combination with the partition size, the offset of the indirectory entry is calculated and the indirectory entry is retrieved — this might cause a second page-fault. Finally, from the indirectory entry, the object offset is extracted and the object contents themselves are faulted-in — this might cause a third page-fault. Both the object and indirectory entry offsets are relative to the partition location, extracted from the PTE.

Notice that, for a single object-fault, the object header/contents need to be accessed twice, as described in the previous section, hence the above process has to be executed twice.

4.9.2 Caching the Partition Location

The *Partition Location Cache* (PLC) is an in-memory data structure that caches some of the contents of the PTE (partition regime, location, and size). It has a fixed number of entries (currently 20, but this can be increased if necessary) and it is organised in a way very similar to the DC, in particular using the same locking mechanism (⇝§4.7). The key of each PLC entry is the PI. Figure 4.17 illustrates a hit on the PLC, that avoids a potential page-fault on the PT.

It can be argued that the introduction of the PLC will have little performance gain. This is because, if a partition is heavily accessed, the corresponding PT page is very likely to be resident anyway. Therefore the page-fault may be avoided and a look-up on the DC is not that much more expensive than a look-up in the PLC. However, the use of the PLC has an additional and very important advantage: it can be used by the garbage collector to determine whether a partition is in-use or not, hence whether the partition can be moved

	SingleBench		MultiBench	
	Number	% of Total	Number	% of Total
Cache hits	990,002	99.0000	3,679,081	99.9750
Secondary hits	0	0.0000	1	0.0000
Cache misses	10,000	1.0000	920	0.0250
Total	1,000,002		3,680,002	
Waited for contents	0	0.0000	1,550	0.0421
Allocations*	10,000		920	
Total searching steps	10,000		920	
Average searching steps	1.00		1.00	

* This is the number of times a PLC entry has been allocated and it it the same to the number of misses. It is replicated for clarity reasons.

Table 4.2: Measurements of PLC Usage

(↝§6). The mechanism for this is outlined below.

- Any thread that is reading information from a partition has to first look-up and pin the corresponding PLC entry (without reading the PTE directly) and keep it pinned while it is accessing its contents. It must be emphasised that this invariant is enforced inside the Sphere calls and no such requirements are imposed on the mutator. Notice also that, as the garbage collector cannot access a partition while it is being updated, the above invariant can be relaxed, if necessary, during updates and the promotion operation.

- When the garbage collector thread needs to move the partition, it blocks any other thread from accessing its PLC entry. When the pin count on it reaches one (this means that only the garbage collector thread is accessing the PLC entry — it is guaranteed that this will eventually happen since no "new" threads can access the PLC entry, and hence repin it, and any "old" ones that are already using it will be allowed to progress and eventually unpin it), the partition can be moved and its PLC entry and PTE updated with the new partition location. Only then can other threads be unblocked and allowed to access the PLC entry, reading the new partition location.

The above mechanism provides a simple and efficient way to guarantee that no threads are accessing a partition so that it can be moved by the garbage collector. Even though it does require some cooperation with the reader threads, this is typical for garbage-collected systems (e.g. the consistent/inconsistent regions of EVM [WG99]).

Finally, some extra functionality that has been added to the PLC is the reference count lock (↝§4.8.5). Each PLC entry contains a mutex that has to be taken when reference counts are updated on the corresponding partition. This lock was added to the PLC entry, rather than the partition table, since it is more efficient to access it in this way. Additionally, the lock is expected to be held for relatively short periods of time (↝§4.10), so it does not impose lengthy pinning requirements on the PLC entry, whereas the other flags described in section 4.8.5 are expected to be held for longer periods of time.

Table 4.2 presents the PLC measurements taken from the two benchmarks introduced in section 4.3 (the table looks very much like table 4.1, which contained the DC measurements —this is because the PLC is using an algorithm very similar to the one presented in section 4.7.3 to ensure MT-safety). The percentage

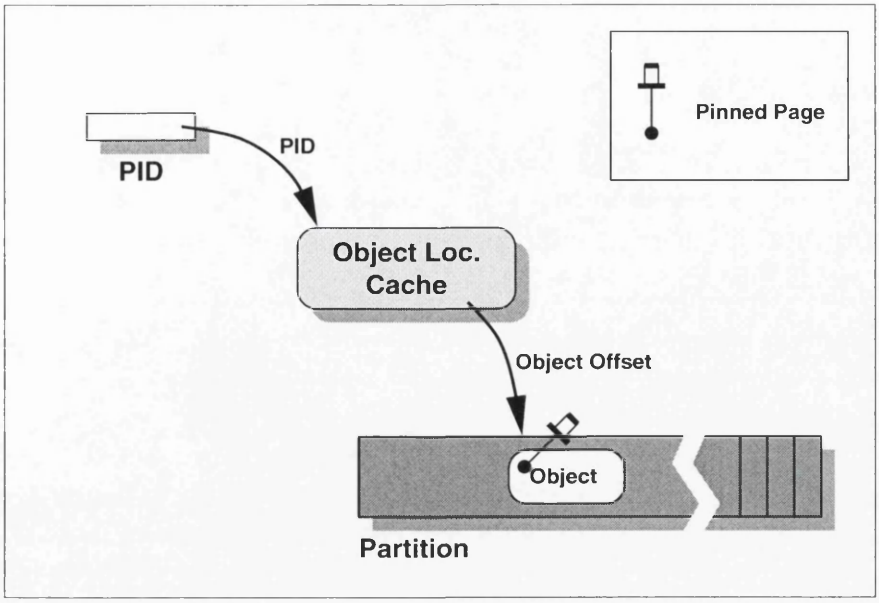


Figure 4.18: The Object Location Cache

of hits is again very high at 99% or over (see section 4.7.4 for an explanation for this), even if there were only 10 PLC entries for 20 partitions in the store. It is worth noticing that in the multi-threaded case there was only 1 secondary hit and the calls that waited for contents very few (under 0.05%).

4.9.3 Caching the Object Location

In an attempt to optimise further the operation of accessing the contents of an object, a second cache was introduced, the *Object Location Cache* (OLC). In order to achieve maximum efficiency, the OLC not only caches the offset of the object inside the partition (this avoids a potential page-fault on the indirectory), but it also keeps the corresponding page pinned and has a direct pointer to the object header on that page. In this way, a cache hit on the OLC can provide immediate access to the object, without any page-faults being necessary[8]. Its operation is illustrated in figure 4.18.

As Sphere assumes that the mutator caches objects anyway (⇝§3.1), the hit-rate of the OLC can only be expected to be around 50% (possibly a cache-miss upon the `sphere_inspectObject` call, but hopefully a cache hit upon the `fetch` operation, as these two calls typically are invoked consecutively). In practice, this has caused the OLC not to provide any performance gains during execution, since the benefit of the very fast cache hit is eliminated by the operation of installing the OLC entry contents (while taking the appropriate locks to ensure MT-safety), upon the initial cache miss.

In order to deal with the above problem, an attempt was made to remove the global OLC and introduce a very small OLC per thread, using thread-specific data (e.g. `pthread_setspecific` and `pthread_getspecific`, if using POSIX threads [LB98]). The advantage of this approach is that, as only one thread can access each OLC, no concurrency control is necessary, therefore the contents of the OLC entries can be installed and accessed very efficiently.

[8]This, of course, does not apply to large objects. In this case, the object header is guaranteed to be accessed with no page-faults, but any subsequent pages may require page-faults.

	No OLC	Thread-Local OLC	Global OLC
SingleBench	22.814 secs	22.7264 secs	22.8892 secs
MultiBench	1,350 lists	1,350 lists	1,260 lists

Table 4.3: Comparing Different Implementations of the OLC

	SingleBench		MultiBench	
	Number	% of Total	Number	% of Total
Cache hits	500,001	50.0000	1,677,631	49.9295
Secondary hits	0	0.0000	322	0.0096
Cache misses	500,001	50.0000	1,682,049	50.0610
Total	1,000,002		3,360,002	
Waited for contents	0	0.0000	3,098	0.0922
Allocations*	500,001		1,682,049	
Total searching steps	500,001		101,574,508	
Average searching steps	1.00		60.39	

* This is the number of times an OLC entry has been allocated and it it the same to the number of misses. It is replicated for clarity reasons.

Table 4.4: Measurements of Global OLC Usage

The three approaches (i.e. no OLC, thread-local OLCs, and a global OLC) were compared using benchmarks. Having no OLC is the simplest of the three. Having a thread-local one is the most complex, since the mutator has to register threads with Sphere so that the thread-specific data can be set up, and has to notify Sphere when a thread terminates, so that the OLC can be de-allocated, etc. Consistently, the global OLC performed worse in both the single-threaded and multi-threaded cases. Additionally, the thread-local OLC performed the same or marginally better than when not using the OLC, in the single-threaded case, and the same in the multi-threaded case.

The results for the two benchmarks introduced in section 4.3 are presented in table 4.3 and follow the trend mentioned above. Additionally, table 4.4 contains measurements for the global OLC. In this case the number of entries in the global OLC was set to 20, 50% of the number of threads used in MultiBench. As expected, the hit rate was 50% for SingleBench and slightly less for MultiBench. However, what is worth noticing is that in the multi-threaded case the average searching steps were over 60 (three times the length of the OLC!). This was caused by all the OLC entries being pinned and some threads having to keep iterating over them, as there were more threads than OLC entries, until one was unpinned.

Based on the above, it was decided to remove the OLC entirely from Sphere as the only performance gain that it could provide was marginal and not worth the extra complexity that it introduced.

4.10 The Reference Count Update Buffer

As illustrated in section 3.7.2, every object has an associated *Reference Count* (RC) that represents the number of cross-partition references pointing to it. This RC, which resides on the object's indirectory entry

(⇝§4.5.3), is updated when one of the following three events occurs.

- **Object Allocation** — When a new object is allocated in the store, it might contain references to objects in other partitions, hence their RCs will have to be increased.
- **Object Updates** — Updates to reference fields might remove and/or create references to objects in other partitions, therefore they might cause their RCs to be updated.
- **Object Reclamation** — If an object points to objects in other partitions and it is reclaimed by the garbage collector, their RCs will have to be decreased.

Applying the RC updates for every such event can be very expensive as pages on other partitions, apart from the one that is being updated, need to be fetched and a lock needs to be taken to ensure that the RC update is MT-safe (⇝§4.8.5).

In order to optimise this operation, it was decided to group RC updates and the *Reference Count Update Buffer* (RCUB) was introduced. Every RC update, rather than being applied directly to the corresponding indirectory entry, is noted on the RCUB. The only pieces of information needed are the PID of the object whose RC is being updated and whether it is being increased or decreased by one. This information can conveniently fit in a word of the same width as the PID (currently 32 bits), since all Sphere PIDs have their least-significant bit always set to 1, to be able to distinguish them from memory addresses (⇝§3.7.1). This gives the opportunity to add a PID of an object to the RCUB as it is, when its RC needs to be increased, or after having set its least-significant bit to 0, when its RC needs to be decreased. When the PID is retrieved from the RCUB, its least-significant bit is examined, determining whether its RC should be increased or decreased, and then set to 1 so that the PID is "re-created" before being used.

The RCUB is organised as a fixed-sized array of PIDs (currently its length is arbitrarily set to 1K elements, however experimental results have shown that larger RCUBs can provide better promotion performance, ⇝§5.3.7) and there is one RCUB per history. When a new history is created, an RCUB is allocated for it and all RC updates of that history will be applied through its RCUB and never directly. Every time the RC of an object needs to be updated, its PID is appended to the array, using the optimisation mentioned above. The RCUB is flushed (i.e. all updates cached in it are applied to the objects) when either the array is full or the history needs to be committed. When the RCUB is flushed, the array is sorted according to the values of the PIDs (the Unix library routine `qsort`, which implements the QuickSort algorithm [Sed88], is used for this). The comparison of the PIDs is implemented in a way that all entries corresponding to the same partition are grouped together and are ordered according to the position of their indirectory entry, and all entries corresponding to the same object are adjacent.

After the array has been sorted, it is scanned linearly. All the entries corresponding to the same object are discovered (this is easily done as they are guaranteed to be adjacent) and the sum of the RC updates for that object is calculated. If this is zero, no RC update is necessary for that object as the ones noted in the RCUB cancel each other out. If it is not zero, the appropriate page is fetched and the update on the object's RC is applied. The way the entries have been sorted ensures that such pages are fetched linearly inside a partition in an attempt to optimise I/O traffic and minimise disk head movement. Furthermore, all RC updates on the same partition take place consecutively, hence the lock for each partition that ensures MT-safety (⇝§4.8.5) needs to be taken and released at most once per RCUB flush. Finally, the grouping of the RC updates allows for a single log record, corresponding to all the RC updates of a single RCUB flush, to be easily composed and written to the log.

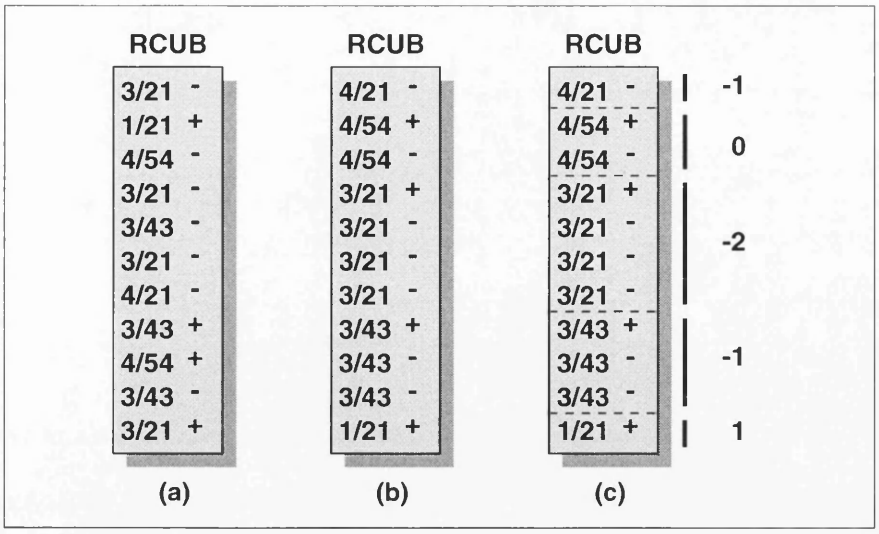


Figure 4.19: The Reference Count Update Buffer

The operation of the RCUB is illustrated in figure 4.19. For simplicity reasons, the figure illustrates PIDs as PI/indirectory index pairs and a + or - represents whether the corresponding RC should be increased or decreased, respectively. Step **A** illustrates the point when the the RCUB is about to be flushed. At this point, it contains 11 entries. Step **B** illustrates the RCUB after it has been sorted. Notice that all entries corresponding to the same partition are adjacent and they have also been sorted according to their indirectory index *backwards*. The latter is because the indirectory grows backwards in the partition and hence, if the entries are sorted backwards, the pages will be accessed in a forward order inside the partition. Finally, **C** illustrates the summing up of the RC updates.
The sensitivity of the RCUB size to the performance of large promotion operations is explored in section 5.3.7.

4.11 Descriptors

This section deals with the handling of descriptors (⇝§3.5.3). Section 4.11.1 describes how descriptors are organised inside a partition and section 4.11.2 outlines the mechanism employed to cache descriptor locations in memory. It must be emphasised that even though these facilities are included in the core of Sphere, the layout and handling of the descriptor objects is the responsibility of the store customisation. In fact, descriptors have to be implemented as an object kind (⇝§3.5.2) and, if necessary, different kinds of descriptors can co-exist in the same Sphere store.

4.11.1 The Descriptor Table

As described in section 3.5.3, a descriptor is allocated in a partition when the first object that depends on it (e.g. an instance of the class the descriptor corresponds to) is allocated in the same partition. Any other such objects that will be subsequently allocated in that partition will point to the same descriptor. Hence an efficient look-up mechanism to discover descriptors inside a partition is necessary. Additionally, the PJama$_1$ evolution mechanism needs to efficiently determine whether a partition contains instances of a class that is being evolved [HAD99]. The invariant assumed is that if a descriptor of a class resides in a partition, then

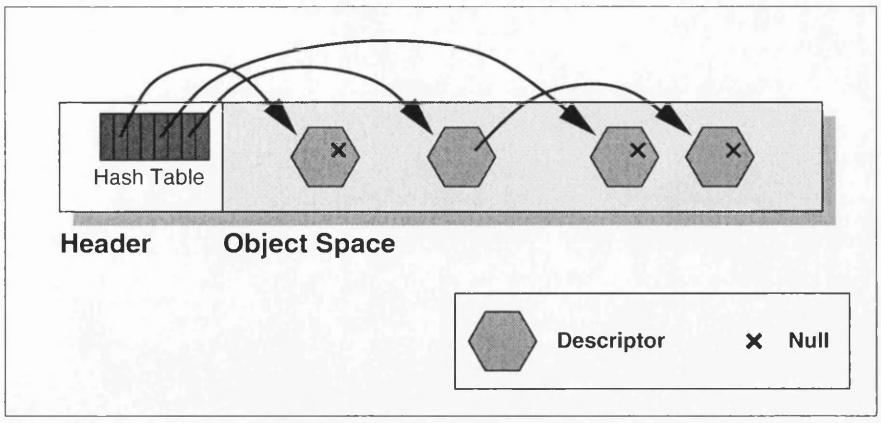


Figure 4.20: Descriptor Organisation Inside a Partition

instances of that class also reside in the that partition (otherwise, the descriptor would have been garbage collected). Given this, an efficient descriptor look-up can also avoid scanning the partition looking for instances that do not exist [HAD99].

The descriptors are organised inside each partition into a linked hash table, called the *Descriptor Table* (DT). The table itself is placed in the partition header (not in all partitions, but in ones that are expected to store descriptors). It was decided to impose the following two restrictions: *(i)* the info field (⇝§4.5.4) of each descriptor is used for linking purposes and *(ii)* the first 32-bit field of the descriptor contents contains the descriptor's key, which is used for look-up and insertion purposes (in the case of PEVM, the key of a descriptor is the PID of the object representing the corresponding Java class). This provided a simple way for the DT management, which is part of Sphere's core, to interact with the descriptor implementation, which is part of Sphere's customisation.

Descriptors in Sphere are first-class objects and are given a PID, even though they are not typically faulted-in explicitly. The DT is using the descriptor PIDs for linking purposes, rather than a more direct addressing scheme (e.g. offsets inside the partition). This allows the implementation of the garbage collector to be kept unchanged and not to have to treat descriptors specially when discovering references in them.

Finally, it is worth pointing out that, at least in the case of PEVM, look-ups on the DT do not take place upon object-faulting, as objects have a direct reference to their corresponding descriptor. Look-ups are only necessary during object allocation, to determine whether the required descriptor is already in the partition or it has to be allocated.

4.11.2 The Descriptor Location Cache

As described in the previous section, when a new object that requires a descriptor is about to be allocated in a partition, a look-up on the DT yields whether the descriptor has already been allocated in that partition or not. However, consider that the promotion operation that allocates new objects in a Sphere store, described in detail in chapter 5, is split into two phases: first allocations (creation of all indirectory entries) and then writes (installation of all object contents). During the allocation phase, look-ups on the DT are necessary. However, if any descriptors are allocated then their contents will be installed during the next phase (writes). It follows that, during the allocation phase, the descriptor chains of the DT are not properly updated, there-

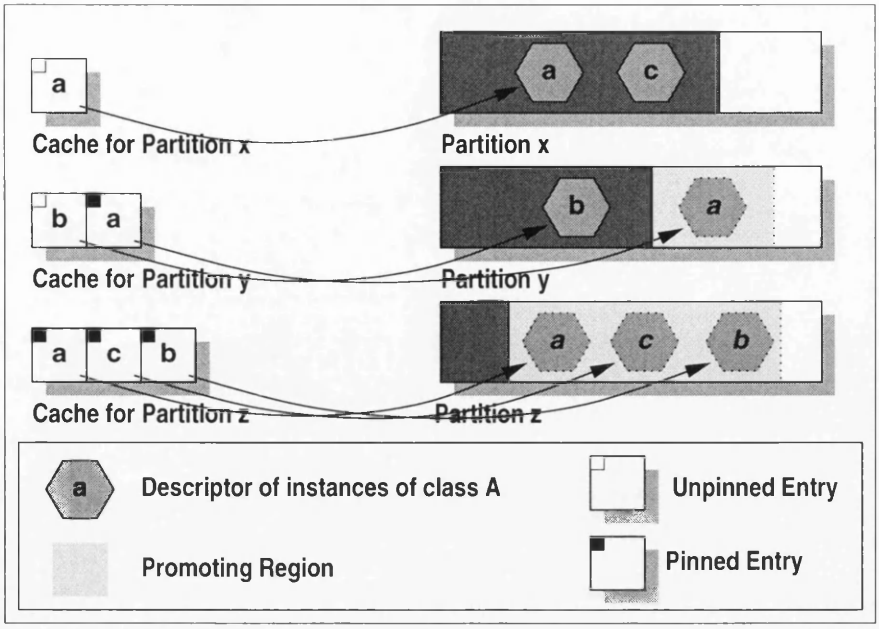


Figure 4.21: The Descriptor Location Cache

fore look-ups on the DT will yield false information (and might in fact find the DT in an inconsistent state).

To deal with this problem, it was decided to introduce the *Descriptor Location Cache* (DLC). This is an in-memory data structure that keeps track of all the newly-allocated descriptors. Additionally, it caches information from the DTs, in order to avoid DT look-ups and hence potential page-faults. So, given a partition and a descriptor key (the PID of the class object in PEVM) the DLC yields whether the corresponding descriptor has already been allocated in that partition and, if yes, also yields its PID.

The operation of the DLC is illustrated in figure 4.21. Notice that not all descriptors in the store have corresponding DLC entries (e.g. the descriptor for class **c** in partition **x**). Also, notice that all newly-promoted descriptors have their DLC entries pinned, so that they cannot be evicted as this is the only place where their PIDs can be discovered. They will be kept pinned until the histories, in terms of which they were promoted, are committed.

At present there is no eviction policy for the DLC. This will be investigated in the near future.

4.12 Concurrency Considerations

Sphere has been implemented to operate in a multi-threaded environment. All the calls and operations of the Sphere public API (⇝§A) are MT-safe, apart from a few that inherently do not need to be (e.g. `sphere_open` ⇝§A.2.3 and `sphere_close` ⇝§A.2.4). This has been achieved by introducing MT-safety when necessary to all Sphere components, as described in most sections of this chapter. The locking mechanism applied to each component was chosen as a trade-off between simplicity and efficiency. For example, the DC is likely to be a "hotspot"; therefore a complicated locking scheme that attempts to maximise concurrency was implemented (⇝§4.7). Alternatively, updates on the PT are likely to be relatively infrequent, therefore a global lock was used instead (⇝§4.8).

The mutator can totally rely on the MT-safety features of Sphere in order to use it correctly. The main exception to this[9], as stated in the requirements in section 3.2, is that the mutator should provide serialisation of updates to the same object by different threads.

4.12.1 Thread Scalability Measurements

Another aspect of Sphere that was measured was its scalability when a large number of threads operate over it. The throughput of MultiBench is plotted in figure 4.22 when the thread count is varied between 50 and 1,500. The figure shows that the benchmark throughput increases linearly when up to 650 threads are used and it degrades after that, also having a few oscillations. The fact that a linear increase in throughput is observed when more threads are added might seem strange, especially as the machine used only has four CPUs. The reason for this is that each store page contains node objects that belong to several lists. When a page is faulted in by one of the reader threads, other threads can also take advantage of it. As all threads start executing at the same time, they move along the lists at roughly the same speed. Therefore a page fetch can allow a number of threads to progress and when a page is evicted any threads that needed to read objects on it have already done so. Even though this is an unlikely situation in practice, and is in fact caused by the synthetic nature of the benchmark, it illustrates that Sphere can take full advantage of best-case scenarios and that no threads are starved or affected by bottlenecks in its MT-safe mechanisms.

Some of the reasons for the throughput degradation when more 650 threads were used can be identified in the following three figures. Figure 4.23 plots the percentage of cache hits for the DC and PLC for the above benchmark run. This increases up to the 650 thread mark then degrades after that. Notice that the shape of both lines roughly follow the one in figure 4.22.

Figure 4.24 plots the percentage of calls that waited for contents for the same benchmark run. This remains consistently low, in the case of the PLC. However, in the case of the DC it steadily improves up to the 650 thread mark and then degrades after that, reaching 45% in the worst case. Interestingly, some similarities between the peaks and troughs of this line and the one in figure 4.22 do exist.

Finally, figure 4.25 plot the average searching steps for the same benchmark. This remains consistently low, in the case of the DC. However, it degrades dramatically in the case of the PLC after the 800 thread mark, reaching a maximum of around 100,000. As less is better in this graph, some of its peaks match troughs in figure 4.22 and vice-versa.

The three figures above indicate that when more than 650 threads were used, the DC became congested with a lot of threads blocked waiting for page contents to be installed. This prevented them from progressing, compared to other threads, and as a result *(i)* a larger number of pages and partitions were touched at any given time and *(ii)* the hit ratio of the DC and PLC was decreased. It also eventually caused the PLC to congest after the 800 thread mark and dramatically increased the average searching steps, as threads waited for PLC entries to be freed.

It is worth pointing out that the throughput degradation for this benchmark could be dealt with if necessary by appropriately tuning the size of the DC and PLC. These were kept low on purpose (25% of the store size and 50% of the number of partitions respectively) in an attempt to force cache misses and demonstrate

[9]There are in fact a few more, but they are mostly obvious, e.g. no other operation should be called before `sphere_open` has completed execution.

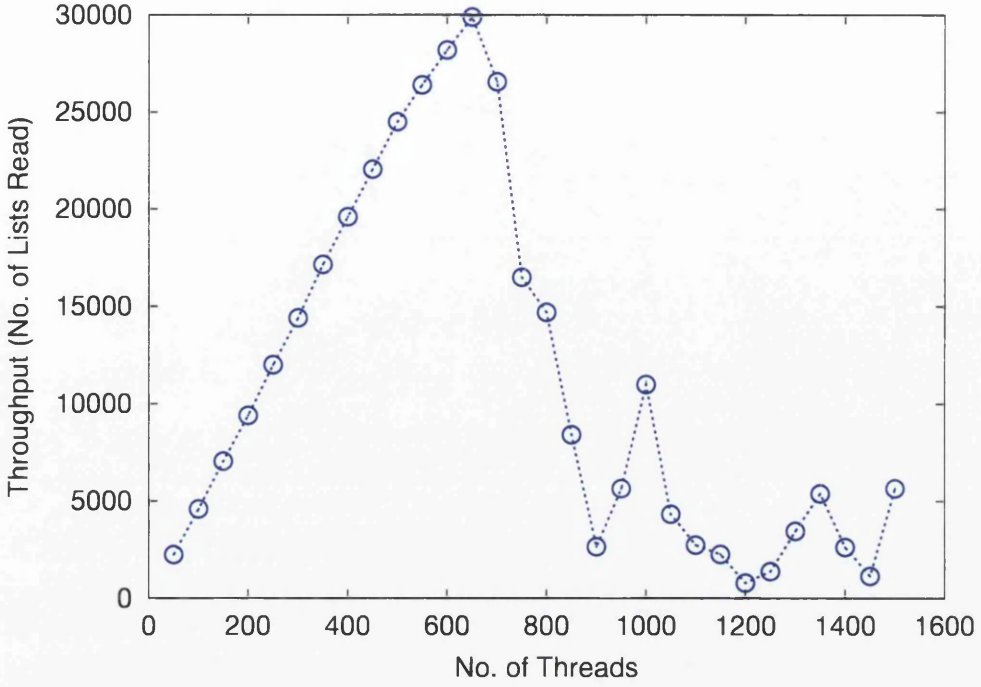


Figure 4.22: Thread Scalability of MultiBench — Throughput

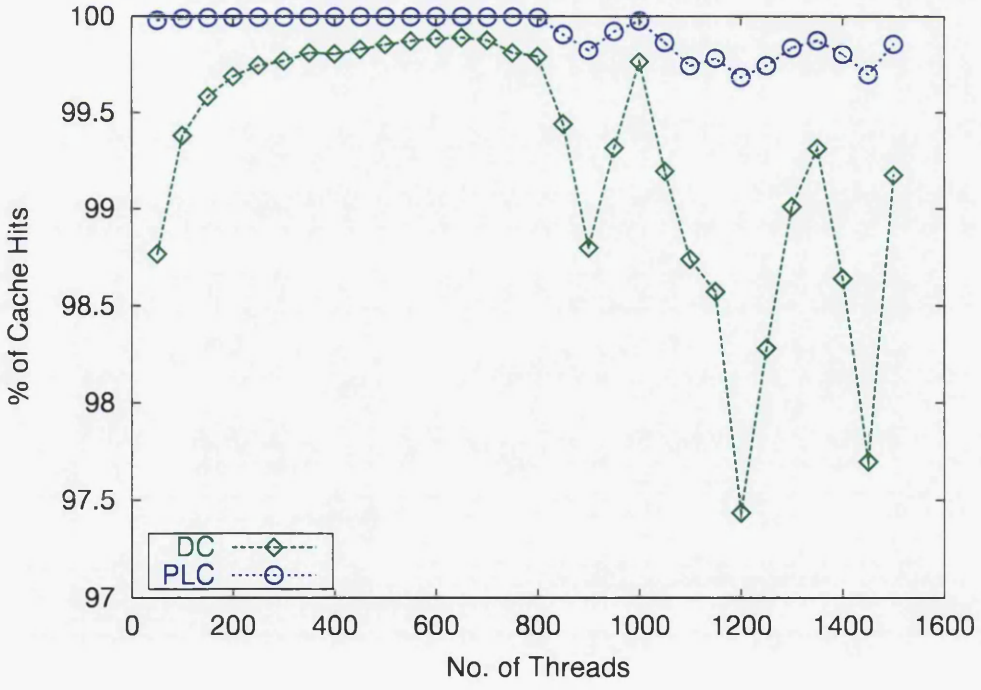


Figure 4.23: Thread Scalability of MultiBench — Percentage of DC and PLC Hits

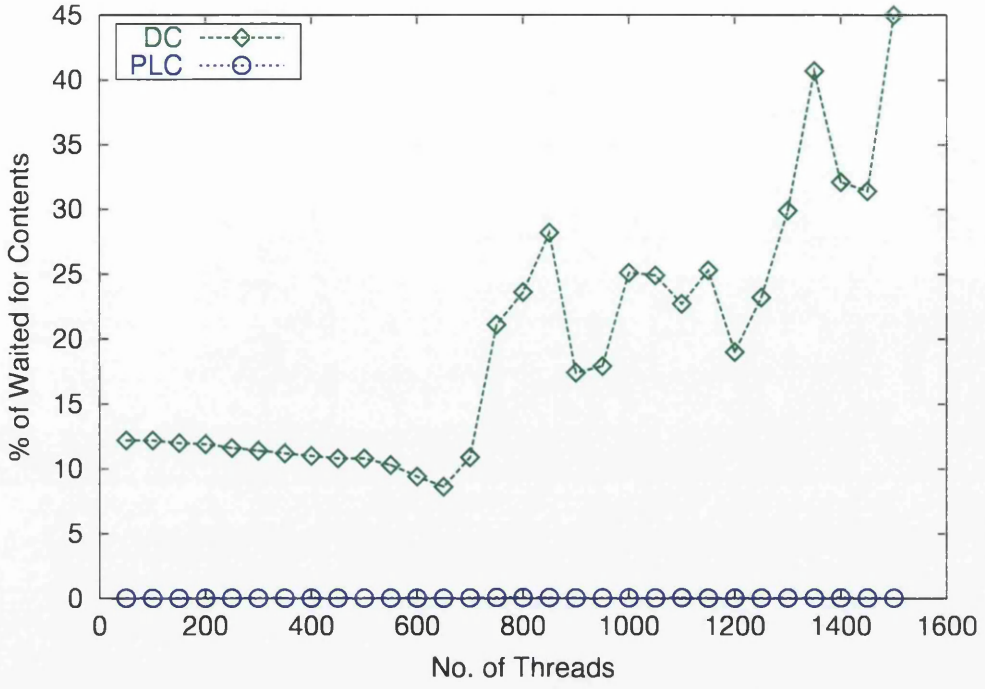


Figure 4.24: Thread Scalability of MultiBench — Percentage of "Waited for Contents" for DC and PLC

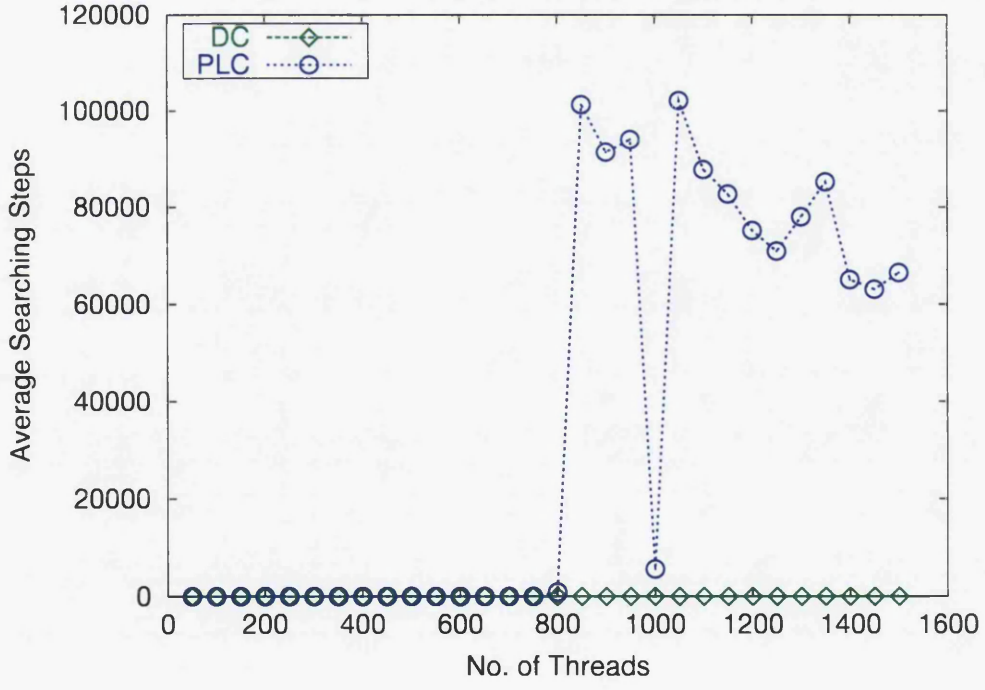


Figure 4.25: Thread Scalability of MultiBench — Average Searching Steps for DC and PLC

their effects. Finally, it is not known whether the thread implementation itself contributed or not to the benchmark's behaviour.

4.13 Customisation

This section describes the implementation of the customisation included in Sphere so that it can serve the PEVM system (⇝§2.5). In particular, section 4.13.1 enumerates the implemented object kinds and section 4.13.2 enumerates the implemented partition regimes and which kinds are stored in them.

4.13.1 Implemented Object Kinds

The five object kinds currently included in Sphere are the following.

❶ **Scalar Kind** — Objects of this kind contain only scalar data. The info field of these objects does not have a specific purpose and can be set to any value chosen by the mutator.

PEVM maps scalar array objects to this kind.

❷ **Reference Kind** — Objects of this kind contain only reference fields. The info field of these objects does not have a specific purpose and can be set to any value chosen by the mutator.

PEVM maps reference array objects to this kind.

❸ **Instance Kind** — Objects of this kind contain both reference and scalar fields. Sphere can deduce their layout from the contents of their corresponding descriptors (⇝§3.5.3 and §4.11). The info field of such an object points to its descriptor. However, this is hidden inside the implementation and the illusion is given to the mutator that the info field points to the corresponding class object, essentially "hiding" from the mutator that the descriptor provides one level of indirection to the class object (⇝§3.5.3).

PEVM maps class instance objects to this kind.

❹ **Instance Descriptor Kind** — Objects of this kind represent descriptors that define the layout of objects of the instance kind (⇝§3.5.3). These are internal to the Sphere implementation and are typically not visible to the mutator.

❺ **Napier Kind**[10] — Objects of this kind contain both reference and scalar fields. In order to be easily identified, all the reference fields are grouped at the beginning of the object and the value of the info field specifies how many of them there are. This kind provides a convenient way of creating objects that contain both reference and scalar fields, without having to introduce descriptors for them.

PEVM maps class objects and some other auxiliary data structures to this kind.

Apart from the descriptors, which are assumed to be small, there are implementations for both small and large objects (⇝§3.4.2) for all of these kinds.

[10]This kind is so named, since the the layout of its objects is the same as the one adopted by the Napier88 system [Bro89], even though it had been previously used in other systems as well, as mentioned in section 3.5.2.

4.13.2 Implemented Partition Regimes

The partition regimes, currently included in Sphere, are the following.

❶ **Compacting Regime** — Partitions of this regime rely on compaction for their free-space management [Jon96]. During garbage collection, the space of all garbage objects is reclaimed by "sliding" all of the live objects towards the beginning of the partition (↝§6.3). In this way, all the free space is gathered towards the end of the partition, which is used linearly to satisfy allocation requests.

PEVM maps the scalar, reference, and Napier kinds to this regime.

❷ **Compacting Regime with Descriptors** — As above, but partitions of this regime also contain the descriptor table on their header (↝§4.11.1), for the correct management of descriptors.

PEVM explicitly maps the instance kind and implicitly maps the descriptor kind to this regime.

There are implementations that support both small and large objects (↝§3.4.2) for both the above regimes.

4.14 Meeting the Requirements

Chapter 3 presented the design of Sphere and this chapter its implementation. The design and implementation decisions, presented in these two chapters, allow Sphere to meet the requirements that were enumerated in section 3.2. Below, it is described how each requirement was met.

- ❑ The ability to extend Sphere with custom-defined object kinds (↝§3.5.2) and the introduction of descriptors (↝§3.5.3) allow a wide range of objects of different types and layouts to be stored by it and Sphere to be able to identify references within them.
- ❑ The introduction of the segment abstraction (↝§3.4.4) allows Sphere to meet its target size of 10GB and use multiple devices, if necessary.
- ❑ The logical addressing scheme adopted in Sphere, i.e. the fact that a PID does not depend on the partition location in the store or the object location in the partition (↝§3.7.1), allows the garbage collector to efficiently move partitions and/or objects without having to patch reference fields. It also allows the evolution of objects, without their PIDs changing (↝§6.7).
- ❑ The adoption of the ARIES algorithm (↝§4.4.3) allows Sphere to overcome the scalability-related problems encountered in the PJama Classic system (↝§2.4.1) and rely on proven and robust technology for its recovery facilities.
- ❑ Sphere is accessed entirely through a well-defined interface, the Sphere public API (↝§4.1 and §A).

4.15 Summary

This chapter described the implementation of Sphere. First, the Sphere component organisation was described and the interaction between its components (↝§4.1). The two types of locks used in the system to ensure MT-safety were presented (↝§4.2), followed by the two benchmarks that were used to obtain measurements of different components of the system (↝§4.3). Then, the recovery mechanism implemented in Sphere to ensure fault-tolerance was outlined (↝§4.4), followed by illustrations of the layouts of the main Sphere structures (segments, partitions, indirectories, and objects, ↝§4.5). The internals of the store

core were then described, in particular the BB management and segment table (⇝§4.6), the disk cache that caches store pages (⇝§4.7), and the partition table that provides one level of indirection to partition locations (⇝§4.8). The location caches were then described (⇝§4.9), followed by the reference count update buffer (⇝§4.10) and the descriptor management (⇝§4.11). MT-safety issues in Sphere were then summarised (⇝§4.12). Finally, the currently implemented object kinds and partition regimes were enumerated (⇝§4.13) and the way the Sphere design and implementation ensure that Sphere meets its requirements was presented (⇝§4.14).

The next chapter deals with the promotion algorithm that has been implemented in Sphere and, based on experimental evidence, demonstrates its scalability and efficiency.

Size matters not. Look at me. Judge me by my size, do you? Hmm? Hmm? And well you should not. For my ally is the Force, and a powerful ally it is. Life creates it, makes it grow. Its energy surrounds us and binds us. Luminous beings are we, not this crude matter.
— **Yoda**, *Jedi Master*

Chapter 5

Object Promotion in Sphere

This chapter describes *Ghosted Allocation*, the bulk object-loading mechanism implemented in Sphere. Section 5.1 gives the motivation behind this work. Section 5.2 describes in detail the *Ghosted Allocation* algorithm and section 5.3 presents the experiments used to demonstrate its efficiency and scalability. Section 5.4 includes the related work and section 5.5 summarises the chapter.

A slightly different version of this chapter has been accepted for publication at the journal *Software — Practice and Experience* [PA00].

5.1 Motivation

The way of populating a POS differs between applications. However, there are many applications that initially import a large amount of persistent data from other sources (legacy databases, text files, etc.) before they operate over it. This is called *bulk loading* and is also referred to as the "Big Inhale" problem [ABD+92]. Some examples of such applications are described below.

- **Geographical Information Systems (GIS)** — GIS systems have to import the geographic data into the POS before they can display it, process it, etc. Typically, such data is read from text files, as is the case for the well-known US Census Bureau's TIGER/Line® data set [US 98, US www], whose size is currently about ten gigabytes.
- **BioInformatics Applications** — Research into microbiology, genetics, evolution, diseases, and pharmaceuticals requires increasingly sophisticated informatics. Large volumes of DNA sequence data[1], protein structures, medical histories, laboratory results, and epidemiological statistics have to be bulk loaded and interrelated. Large indices are also built as they are needed to accelerate searches and comparisons.

[1] For example, the recently mapped human chromosome 22 has approximately 33 megabases annotated with structured information. The complete human genome, which will be mapped within a few years, has approximately 3 gigabases [Dun99, Natwww]. Each base will require approximately 30 bytes of data, once indices and mapping annotations are introduced.

- **Experimental Physics Applications** — Large-scale physics experiments generate large amounts of data that need to be first loaded into a store before it can be processed. An example of this is the new particle accelerator at CERN that will be operational in about five years and is expected to generate about one petabyte of data per year [Ard99a]. An OODB capable of handling this load is currently being sought [Ard99b].

Notice that bulk object-loading may be a recurring event; e.g. each experiment at CERN will generate large amounts of new data and several experiments will be run per day. It follows that applications like those mentioned above require a fast and scalable bulk object-loading operation.

The above requirement was also identified, after years of experience gained by building and using persistent object systems (these include several versions of PS-algol [ACC82], Napier88 [MCC+99], and PJama [AJDS96, ADJ+96, Jor99b]). This chapter presents an algorithm called *Ghosted Allocation* that has been designed for efficient allocation of large numbers of objects in a POS. Its main strengths are that it minimises I/O traffic, optimises the disk access pattern, and does not impose complex requirements on the application that is using the POS. It has been implemented and analysed in the context of Sphere (↝§4 and §3). In particular, as the object-allocation operation is tied to the corresponding application-specific partition regime (↝§3.5.1), all the implementation work for this chapter took place outside the Sphere core.

It is typical in architectures for which Sphere is targetted that memory-resident objects fall into two categories: *persistent*, objects that have already been allocated in the POS and are currently cached in main memory for efficiency reasons, and *transient*, objects that have not yet been allocated in the POS and only reside in main memory. In some systems, this distinction is available at the programming level (e.g. PJama [DA97, PAJ99] and GemStone/J [Gem98b]). In some others, all objects are considered persistent by the programmer, but the above distinction still exists internally as an optimisation to avoid writing to disk very short-lived objects (e.g. Napier88 [Bro89] and PS-algol [ACC82]). Transient objects can eventually become persistent according to the identification mechanism implemented by the system, e.g. explicitly by the programmer or by the principle of reachability [AM95]. This chapter only concentrates in the latter case.

At a checkpoint, all of the cached persistent objects, which have been faulted-in from the POS and updated since the last checkpoint, must have their modified state written back to the POS (various algorithms for identifying this set of objects are discussed elsewhere [DA97, LM99, HC99a, HC99b]). When updates to these modified persistent objects result in a reference to an object that was hitherto transient, that object and all the objects *persistently* reachable from it [AM95, JA99] must also be made persistent. That is, they must be allocated PIDs and space in the POS, copied to the POS (they remain cached in main memory to support continued computation after the checkpoint), and their status changed to persistent, so that updates on them are noted during the intervals to subsequent checkpoints. This transition of objects from the transient to the persistent state will be referred to as *Promotion* (↝§2.4).

As described earlier in this chapter, bursts of data creation have been observed in many applications, such as *bulk loads* of data imported from various sources or the construction of new indexes. In order that a system can cope adequately with these bursts of data creation, it is necessary to devise efficient and scalable promotion algorithms, such as the one described in this chapter.

5.2 The *Ghosted Allocation* Promotion Mechanism

This section provides a detailed description of the *Ghosted Allocation* algorithm. Section 5.2.1 describes a technique to extend a persistent space efficiently and fault-tolerantly. Section 5.2.2 gives an overview of the algorithm and section 5.2.3 describes its operation on a single partition. Section 5.2.4 outlines the unswizzling mechanism employed and section 5.2.5 describes how the algorithm can be extended to deal with promotion to multiple partitions simultaneously. Section 5.2.6 enumerates the requirements imposed on the mutator and section 5.2.7 concludes the section.

5.2.1 Efficient Extension of a Persistent Space

This section describes a technique to atomically extend a persistent space. This technique is required by the *Ghosted Allocation* algorithm and its strength lies in that it requires a minimal amount of log traffic. Figure 5.1 provides the legend for the figures of this and subsequent sections.

Consider the situation in figure 5.2 which shows a space that already spans over two and a half disk pages and a field on the first page, called **Limit**, contains its size. This space must be extended atomically and fault-tolerantly with the extension data illustrated in the figure. One way to achieve this is to write all the extension data to the log, then update the disk pages. If a failure occurs, the contents of the disk pages can be recreated from the information in the log. This approach however has the disadvantage that the extension data is replicated twice, once in the space itself and once in the log, hence the log traffic is proportional to the extension size.

A more efficient approach is illustrated in figure 5.3. Only the information that is written to already-existing pages of the space needs to be logged. This consists of the update to the **Limit** field (on page 0) and any new data added to the previous last page of the space (i.e. page 2). All the new pages (i.e. pages 3–4) can be written to disk without being logged; however, they have to be written synchronously. Figure 5.4 illustrates the state of the space after the extension has been committed. The new pages (i.e. pages 3–4) have been written to the disk and the dirtied "old" pages (i.e. pages 0 and 2) can be written asynchronously to disk sometime in the future, as the two updates on them have been logged.

The above algorithm is guaranteed to be fault-tolerant. If there is a failure *before* the extension is committed, enough information is stored in the log to undo the update to the **Limit** field, rolling back the extension. In this case, any new pages that were written to disk can safely be discarded. If there is a failure *after* the extension is committed, enough information is stored in the log to redo the update to the **Limit** field, as well as recreate the data on the previous last page (i.e. page 2). In this case, the contents of the new pages do not need to be recreated, as they are guaranteed to already reside on disk, due to the synchronous writes.

The strength of this technique lies in the fact that the amount of data required to be written to the log to ensure fault-tolerance is, at most, a single disk page and is not proportional to the size of the extension. Furthermore, all the writes to the new pages of the space can take place linearly in the space and can be performed using transfer units larger than the disk page to improve I/O performance (⇝§5.3.5). The store extension operation in PJama Classic uses a very similar technique [DA97].

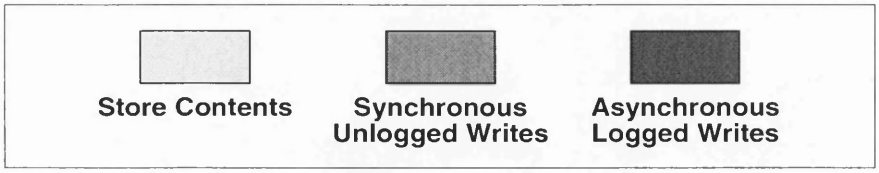


Figure 5.1: Legend

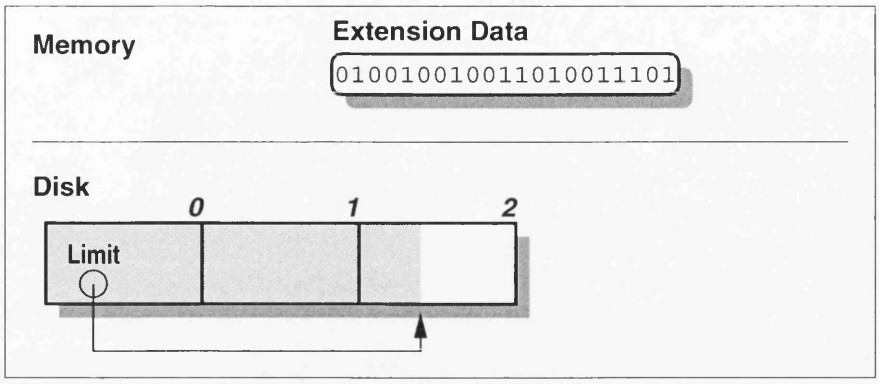


Figure 5.2: Extending a Persistent Space — Initial Configuration

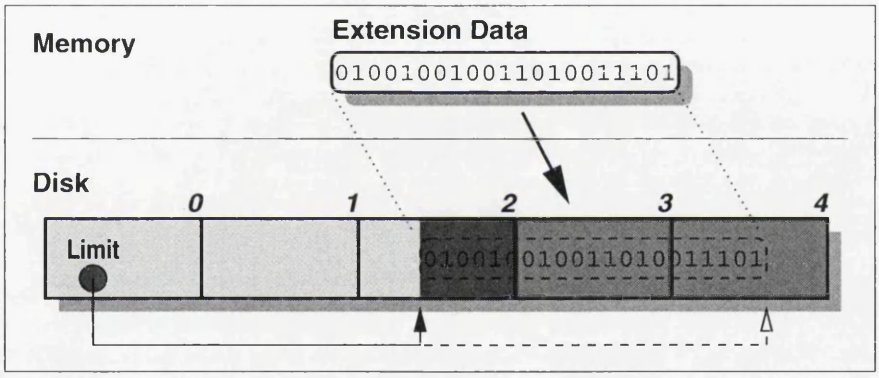


Figure 5.3: Extending a Persistent Space — Performing the Extension

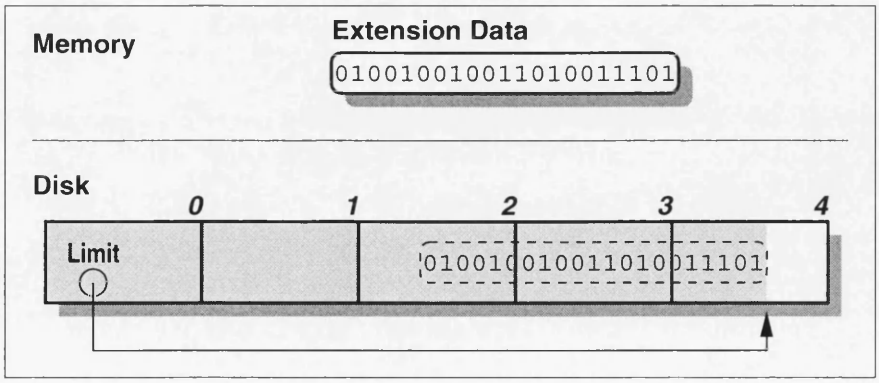


Figure 5.4: Extending a Persistent Space — Committing the Extension

5.2.2 Overview of the *Ghosted Allocation* Algorithm

One of the fundamental assumptions of the *Ghosted Allocation* algorithm is that two operations must be invoked for each object that is allocated in the store. In Sphere these two operations are the following [Pri99b].

- `allocate` — it allocates a new PID and reserves enough space for the object in the store.
- `firstWrite` — it installs the contents of the newly-allocated object in the store.

There are several restrictions on the use of the `allocate` and `firstWrite` calls and the mutator is trusted to comply with them [Pri99b]. These are enumerated below.

- All promotion activity within a single stabilisation must be associated with the same active history (⇝§4.4.1).
- All invocations to the `allocate` call within a single stabilisation should be grouped together. This will be referred to as the *Allocation Phase* of promotion.
- All invocations to the `firstWrite` call within a single stabilisation should be grouped together. This will be referred to as the *Writing Phase* of promotion and must take place after the allocation phase.
- The `allocate` and `firstWrite` calls for the same object should take place in terms of the same history.
- All `firstWrite` calls must take place *in the same order* as the `allocate` calls that allocated the objects. For example, if `allocate` is called on objects **A**, **B**, and **C**, in that order, then `firstWrite` should be called on objects **A**, **B**, and **C**, in that order.
- After an object is allocated with the `allocate` call, the mutator should not cause any other operation to be called on it, apart from `firstWrite`, until the corresponding history has been committed.

The introduction of the above restrictions allows the `allocate` and `firstWrite` calls to be efficiently implemented, as described in detail in section 5.2.3. Section 5.2.6 describes ways of implementing these restrictions efficiently.

5.2.3 Promotion to a Single Partition

This section describes how the *Ghosted Allocation* algorithm allocates objects in a single partition. Figure 5.1 contains the legend for the figures contained in this section. Figure 5.5 illustrates a partition, which will be referred to as partition **PN**, before any new objects are allocated in it. In this example, the partition comprises 8 pages. Page 0 is the partition header and is exclusively reserved for administrative information. The header contains the **OL** and **IL** fields that represent the *object-space limit* and the *indirectory limit* respectively.

When the first `allocate` call is invoked on the partition, its header page is pinned in memory for the entire duration of the allocation phase on that partition. In this way information that needs to be accessed by all `allocate` calls can be included in it. Even though this additional information could have been stored in a main-memory data structure, it seemed natural to include it in the partition header since administrative fields in the header would need to be accessed during promotion anyway. By keeping the partition header pinned

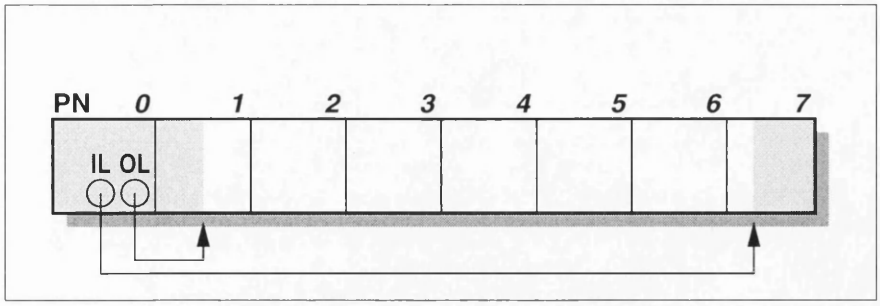


Figure 5.5: *Ghosted Allocation* — Initial Configuration

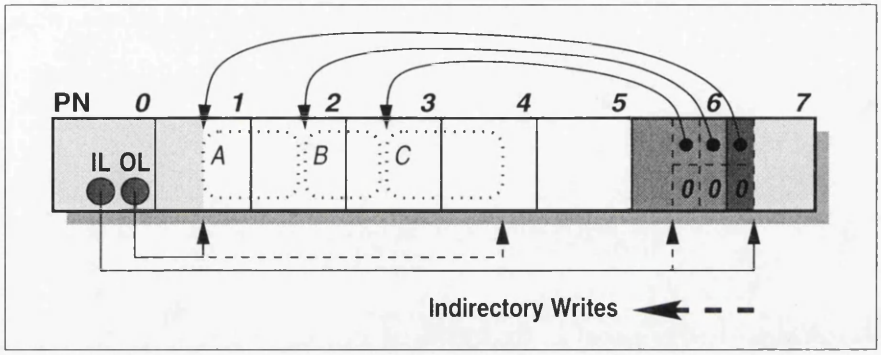


Figure 5.6: *Ghosted Allocation* — End of Allocation Phase

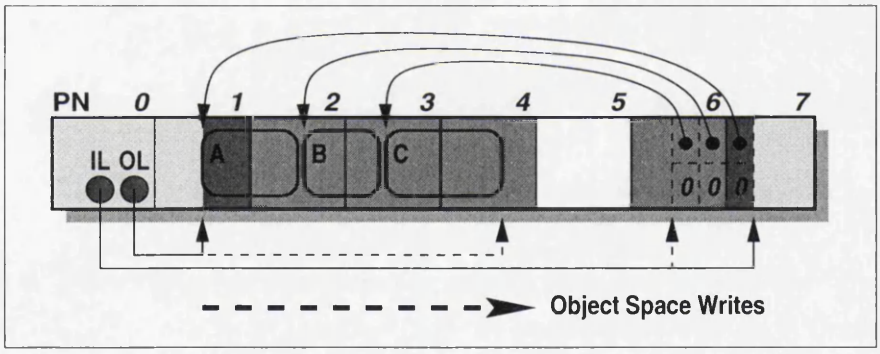


Figure 5.7: *Ghosted Allocation* — End of Writing Phase

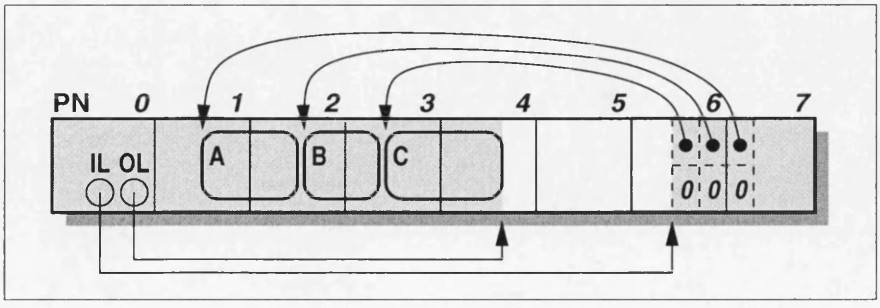


Figure 5.8: *Ghosted Allocation* — Committing the Promotion

ensures that, since it is going to be heavily accessed as objects are allocated in the partition, no unnecessary disk accesses will be needed to refetch it, if evicted.

Every `allocate` call causes space in the partition to be reserved for the corresponding object and an indirectory entry to be allocated by extending the indirectory space[2]. Since the offset of the new object is known, the indirectory entry can be initialised to point to it and its reference count set to 0. However, no writes are performed to the object space during this phase.

It is important to notice that the indirectory entries are allocated linearly. This ensures that all entries on the same page are allocated consecutively, before moving on to the next page. Taking advantage of this, the current indirectory page is pinned until it is full and is only unpinned when it is not required any more. In a manner similar to the partition header, this ensures that this page is not evicted, only to be refetched shortly afterwards.

The indirectory space is extended backwards into the partition using the technique described in section 5.2.1. Only the new entries residing on the last already-existing indirectory page (in the example, page 7) have to be written to the log, along with the update to the **IL** field; all the other entries can be written to disk using synchronous unlogged writes. Given that all new entries are added consecutively to the indirectory, this is easy to arrange.

A partition is full when the indirectory limit reaches the object space limit. When this is detected, a look-up on the partition table takes place for a partition open for allocation (↝§4.8.4). If such a partition does not exist, a new one is created.

Figure 5.6 illustrates the end of the allocation phase on partition **PN**, after objects **A**, **B**, and **C** have been allocated in it, in that order. Both the object space and the indirectory have been extended accordingly, but writes have only taken place on the indirectory pages; the new entries on page 7 have been logged, however the ones on page 6 were written with an unlogged write. Notice that even though the object images have not been written, "ghosts" of the objects have increased the **OL** field, hence the name of the algorithm.

After the allocation phase is complete, the writing phase is initiated on the partition with the first invocation of the `firstWrite` call. This has to first ensure that the header page is fetched and pinned in memory, since it will be accessed throughout the writing phase on this partition (some administrative information is stored in it, e.g. offset of the next object to be written). The fact that the `firstWrite` calls have to visit the objects in the same order as they were allocated, guarantees that all the data in the object space will be written linearly. Consequently, no indirectory accesses are necessary to find the location of the corresponding objects. The extension of the object space also uses the technique described in section 5.2.1, with the only difference that the **OL** field was increased during the allocation phase. Also, keeping the current object space page pinned avoids unnecessary evictions and refetches.

The installation of the contents of an object may increase the reference count of an object in another partition. If this is a newly-allocated object, this can be safely done, even if its contents have not yet been installed, since its indirectory entry will have been initialised during the allocation phase, including setting its reference count to 0.

[2]This is not always true, since there is a free-list of unused indirectory entries that were freed by the garbage collector and may be scattered throughout the indirectory (↝§4.5.3). Here, for simplicity reasons, this is ignored.

Figure 5.7 illustrates the end of the writing phase on partition **PN**. All object contents have been written, the data on page 1 has been logged, and pages 2–4 have been written with unlogged disk writes. Finally, figure 5.8 illustrates the same partition after the history, in terms of which promotion took place, has been committed.

To summarise, the strengths of the above algorithm include minimising the log writes (at most two disk pages per partition), improving the locality of disk writes (most consecutive writes are linear), and avoiding unnecessary evictions and refetches of disk pages (due to the pinning of disk pages). The latter is not optimal since reference count updates might cause an indirectory page, which has already been written and unpinned, to be refetched and updated. However, the RCUB described in section 4.10 ameliorates this refetch cost substantially. Finally, it is worth pointing out that the total pinning requirements are low, up to two pages at any given time (partition header and the current indirectory or object space page that is being operated on).

5.2.4 The Unswizzling Mechanism

The `firstWrite` call, as described in section 5.2.3, accepts a parameter that points to the object contents in memory and writes them to disk. Obviously, the object must be in store format before it can be written to the store. The transformation from memory format to store format is called *unswizzling*. It includes changing object references to PIDs and setting transient fields to default values [PAJ99, JA98]. There are two obvious ways in which the mutator can achieve this.

❶ *Transform the object to store format in-place, copy its contents to a disk buffer, and then transform it back to memory format.*

This has the disadvantage that during this operation the object in memory cannot be accessed concurrently by another thread since it is in the wrong format. A more subtle point is that, even though it is typically easy to translate PIDs to memory addresses (e.g. the resident-object table in PJama supports this [DA97]; a similar data structure exists in the Napier88 system [Bro89]), it is impossible to translate Java transient fields back to their original value on transforming back to memory format, as they have been set to the same default value [PAJ99]. Because of this, the values of such fields would have to be stored somewhere before unswizzling takes place. This in fact is the solution adopted by PJama Classic.

❷ *Copy the object to an intermediate buffer and apply the necessary transformations there before writing the contents of the buffer to disk.*

Even though this deals with the two disadvantages of the previous solution, it does so by forcing the entire object to be copied one extra time and it requires the intermediate buffer to be reserved. This imposes extra space and cycle requirements on main memory.

A third solution that does not have the shortcomings of the two solutions above was adopted in Sphere. Notice that, when an object is written to disk, it first has to be copied to a buffer in the disk cache. This presents the opportunity to perform the necessary transformations in the disk-cache buffer, without disturbing the original image of the object. An additional requirement is to allow this operation to take place without breaking the well-defined interface between the store layer and the mutator and without exposing the disk-cache implementation to the mutator.

For this to be achieved an up-call to the mutator, called an *Unswizzling Callback*, has been introduced in Sphere. It is called every time an object needs to be unswizzled in the disk cache, either due to a `firstWrite`

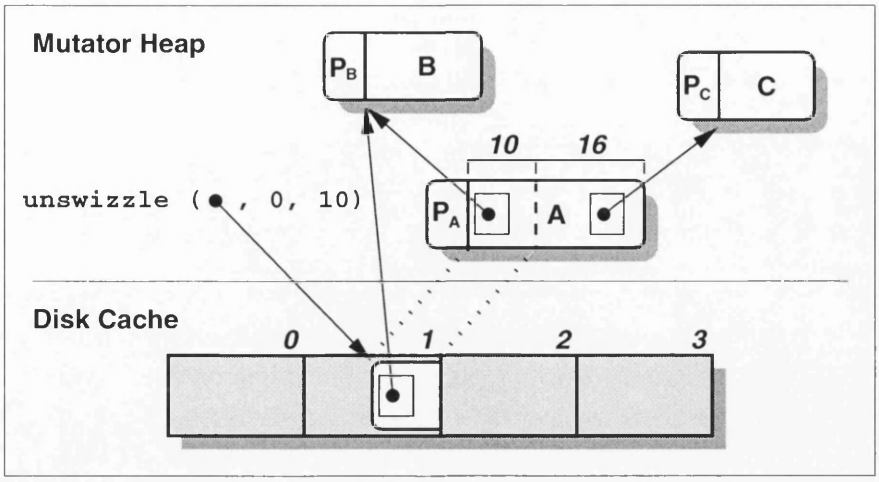


Figure 5.9: Unswizzling an Object — First Page

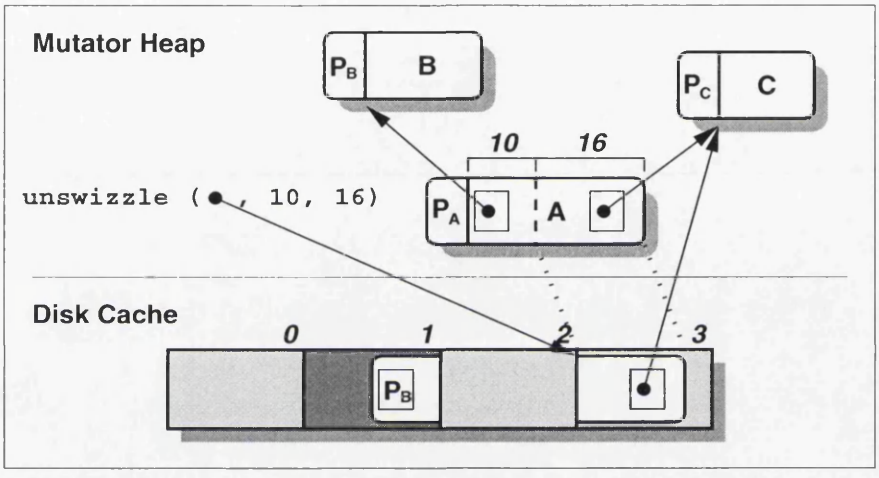


Figure 5.10: Unswizzling an Object — Second Page

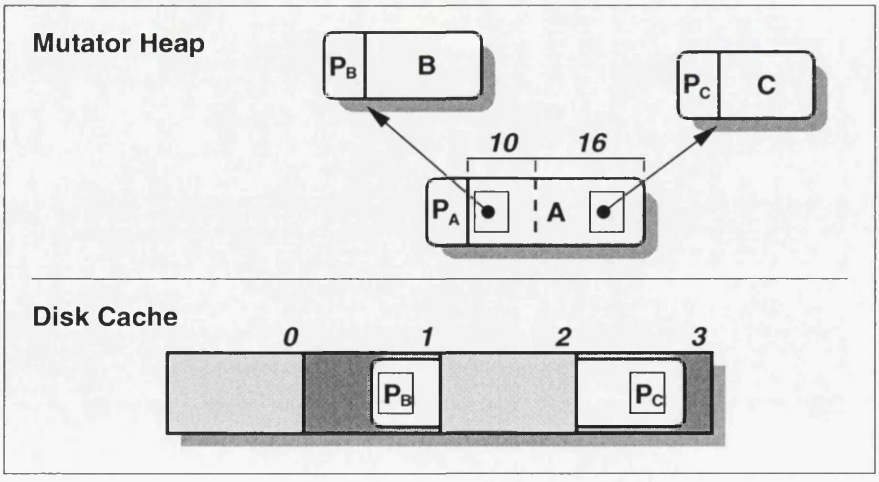


Figure 5.11: Unswizzling an Object — Finished

call or due to one of the two range update calls (`updateAll` and `updateRange`, ⇝§A). If the object spans several pages in the store, the unswizzling callback will be called once per sub-range of the object that resides on each page. The parameters to the up-call are a reference into the disk-cache buffer, where the object has been copied, and the size and offset of the object sub-range that is being unswizzled. Obviously, during the operation of the callback, Sphere keeps the corresponding buffer pinned in the disk cache to prevent it from being evicted.

A concrete example of the operation of the unswizzling callback is given here. In figure 5.9, object **A** is being written to the store with a `firstWrite` call. It points to objects **B** and **C**. The PIDs of the objects have already been allocated and they have been stored in the object headers (in the figure, they are denoted by $\mathbf{P}_A$, $\mathbf{P}_B$, and $\mathbf{P}_C$ respectively).

Object **A** spans two pages. In figure 5.9, the first page is being written and the first part of the object has been copied to disk-cache buffer 1. Next, the unswizzling callback is called on that buffer, with parameters being the address where the beginning of the object resides in the buffer, and the offset (0) and size (10) of the object sub-range being unswizzled.

Figure 5.10 illustrates the second sub-range of object **A** being written to the store. It has been copied to disk-cache buffer 3 (buffer 2 is assumed to be in use for some other purpose) and the corresponding unswizzling callback is being called. Notice that the reference to object **B** on the first object sub-range was overwritten by the previous callback with $\mathbf{P}_B$ and the page may now be written to disk.

Finally, figure 5.11 illustrates the state of the system after the `firstWrite` call has finished writing object **A** to the store. The second callback overwrote the reference to object **C** with $\mathbf{P}_C$, on disk-cache buffer 3, and the page is ready to be written to disk, once any other new or updated objects in it have been treated similarly.

In Sphere, the `firstWrite` call, as well as the `updateAll` and `updateRange` calls, provide extra flexibility by allowing the object contents not to be copied before the callback is called (⇝§A). This leaves the callback responsible for completely filling in the contents of the object. This might be desirable, for example, when writing objects that contain only references, to avoid copying the memory references to the disk-cache buffer and then immediately overwriting them with the corresponding PIDs.

5.2.5 Multiple Partition Considerations

One of the most important facilities of Sphere, as mentioned in sections 3.5.1 and 3.8.3, is that it allows different partitions to implement different free-space management techniques in order to efficiently accommodate objects of different size and behaviour. This implies that, during promotion, consecutively allocated objects might be allocated in different partitions. It is easy to see how the *Ghosted Allocation* algorithm, described in section 5.2.3, can be extended to deal with this.

The presence of multiple partitions is completely hidden from the mutator, which only has to ensure that the order of all `allocate` and `firstWrite` calls is the same. Upon an `allocate` request, the store decides which partition will hold the object, reserves space in it, as described earlier, and returns to the mutator the newly-allocated PID. All `firstWrite` calls take as a parameter the PID of the object. From this, the partition where the object was allocated can be deduced (⇝§3.7.1) and the object contents installed in that partition. According to this, keeping the overall order of the requests the same can also ensure that the requests will appear in the same order in each partition.

A potential complication that arises from the requirement of allocating to multiple partitions is that it might cause the disk accesses to be scattered around the disk, violating the good locality of writes that the *Ghosted Allocation* algorithm allows. There are two possible solutions to this. The first is to ensure that writes take place asynchronously and are opportunistically scheduled to improve locality. This might complicate the committing of the promotion operation since all pages that should be written with unlogged writes should be written to the store before the log can be committed.

A second way to deal with the above problem is to ensure that partitions that are allocating objects at the same time reside on different physical disks. In this way the write operations on one disk will not affect the others. In fact, this also has the potential to optimise the fetching of large data structures (⇝§3.5.4).

5.2.6 Requirements for the Mutator

Ensuring that the order of the `firstWrite` calls is the same as that of the `allocate` calls might, at first, seem like a heavy requirement for the mutator. However, in practice it is easily achieved, imposing minimum or even no changes to the mutator's persistent heap implementation.

In orthogonal persistent systems, the identification of newly-persistent objects is typically achieved by a recursive object-scanning phase, similar to the marking phase of a garbage collector [Jon96]. This phase starts from all the modified persistent objects that contain references to transient objects[3]. The `allocate` requests can be issued during this phase, as objects are being visited.

One simple way to ensure that the `firstWrite` calls are invoked in the same order as the `allocate` calls is to simply repeat the recursive object-scanning phase for a second time, invoking `firstWrite` instead of `allocate`. For this to operate correctly, the object contents should not change between the phases (at least the contents of the reference fields in them). This however is a more general requirement for the correct operation of any promotion algorithm.

If in-memory object updates were allowed between the allocation and writing phases, the latter might visit some objects that had not been allocated PIDs, since they were not persistently reachable during the allocation phase. Such an event would be disastrous. Furthermore, relying on a repetition of the recursive object-scanning phase can be expensive, because of the book-keeping information that such a phase requires. This technique can be easily adopted in any system and its cost is expected to be low, compared to the cost of the disk accesses (this in fact is the case, ⇝§5.3).

Another way to ensure that the objects are traversed in the same order during the allocation and writing phase applies to systems which, during promotion, need to evacuate the newly-promoted objects from a transient heap to a persistent object cache (e.g. PJama Classic). This evacuation can take place during the allocation phase and objects can be copied to a contiguous area in the object cache. During the writing phase, a forward sweep over this contiguous area will ensure the correct order of the `firstWrite` requests.

Finally, an alternative scheme is to link the objects into a linked list during the allocation phase and iterate over this list during the writing phase. This assumes that there is a temporarily available field per object to

[3] A write-barrier must be employed by the mutator to detect write operations on persistent objects. More details on alternative implementations for such a write-barrier can be found elsewhere [HBM93, DA97, LM99, HC99a, HC99b].

be used for linking. In PJama Classic, for example, there is a field in each object handle [DA97] that could be used for this purpose.

5.2.7 Summary

This section has described the operation of *Ghosted Allocation*, an efficient object promotion mechanism. Its strengths include the following.

- There is a well-defined interface between the store and the mutator and the restrictions imposed on the mutator can be achieved easily.
- The algorithm tries to optimise I/O accesses. In particular, it minimises logged writes in favour of safe unlogged writes and attempts to improve their locality.
- It minimises traffic to the log while guaranteeing fault-tolerance. Provided that the newly-allocated indirectory entries extend the indirectory (rather than being allocated from the indirectory free-list), any promotion to a given partition is guaranteed to only generate log traffic equal to two disk pages, in the worst case. Also, the RCUB contributes in generating compact log records for grouped reference count updates.
- It attempts to avoid unnecessary page-eviction and refetching by keeping pages that it is using pinned in memory and not touching them after it has finished operating over them. This does not however apply to indirectory pages, which might be refetched so that reference count updates can be applied.
- The page-pinning requirements of the algorithm are two pages per partition that is currently being operated on (i.e. touched by the allocation or writing phase). The number of partitions that are being operated on at any time is expected to be low (around four or five maximum, since up to that many different free-space management schemes are envisaged to co-exist in the same store).

However, there is a price to pay for the above advantages. The implementation of the *Ghosted Allocation* algorithm was not straightforward and its maintenance is equally complicated. Furthermore, if it operates over partially-populated partitions, its performance is expected to degrade. This degradation is explored by some of the experiments included in section 5.3.

5.3 Measurements

This section presents the experiments that were conducted to evaluate the *Ghosted Allocation* promotion algorithm. Sections 5.3.1 and 5.3.2 describe the experimental platform and mutator load that was used. Section 5.3.3 gives an overview of the experiments. Section 5.3.4 presents a performance comparison between *Ghosted Allocation* and an alternative promotion algorithm. Section 5.3.5 explores the effect the transfer unit size has on the performance of *Ghosted Allocation* and section 5.3.6 measures the sensitivity of the algorithm to the store's disk-cache size. Section 5.3.7 attempts to measure the overhead that reference count updates impose on the system and section 5.3.8 explores the scalability of *Ghosted Allocation*, when dealing with promotion sizes up to 1GB. Section 5.3.9 concludes the section.

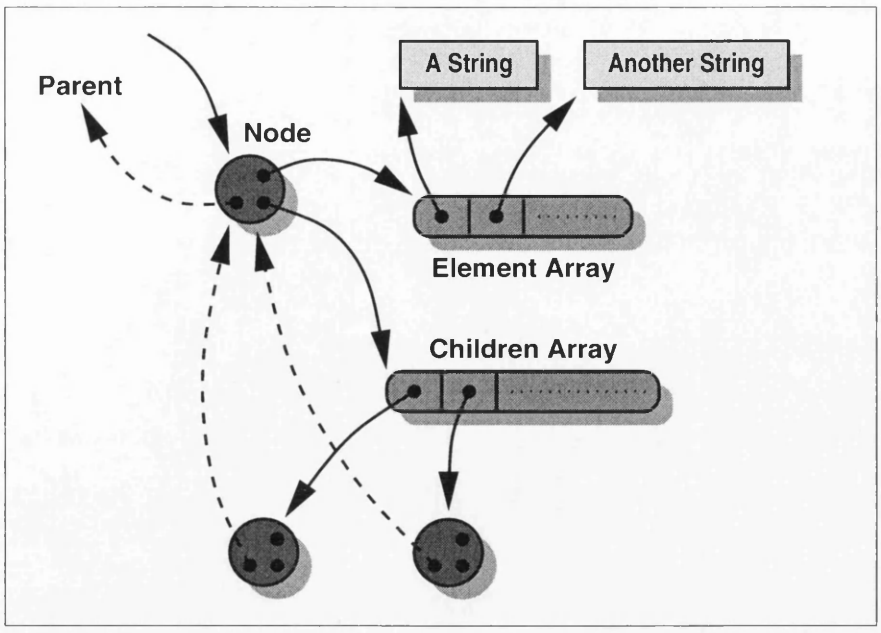


Figure 5.12: B-tree Bode Structure

5.3.1 The Experimental Platform

A very simple custom mutator was used to conduct the experiments presented in this section, instead of the full-fledged PJama interpreter. This allowed most aspects of the system to be measured and reasoned about more easily than if a larger and more complex system had been used.

This mutator consists of a simple heap where objects are allocated and manipulated. Each object has an 8-byte header that contains some flags, the type of the object (so the heap could find the reference fields in it), and a slot for its PID. Once the heap is populated, all objects reachable from a designated persistent root can be promoted to a Sphere store through a single promotion operation. This heap is very simple, i.e. it does not support object de-allocation or update tracking. However, it was sufficient for the experiments presented here, as the only facility the promotion operation requires is an object traversal.

All experiments were run on the machine described in section 4.3 and the Sphere configuration included a single 1GB segment and a 2GB log. The specification of the machine might seem excessive, especially because the multiple CPUs were not really necessary. However, it was the only machine available with enough main memory to ensure that all the heaps used for the experiments (up to 864MB) were entirely memory resident, along with the Sphere data structures.

5.3.2 The Experimental Load

The load used for the experiments was the promotion of a single B-tree [Sed88, TLA90], implemented on top of the heap described in the previous section. The order of the B-tree was arbitrarily set to 19 (i.e. each node has up to 18 data elements and up to 19 children). The B-tree was populated from a text file, with each line of the file providing a single data element that was inserted in the B-tree.

The structure of each B-tree node is illustrated in figure 5.12. Each node is made up of three objects: the

Data File	File Size	Lines	Promotion Size*	Traversal Time
d20	20MB	454,507	37MB	0.35 secs
d40	40MB	900,571	75MB	0.70 secs
d50	50MB	1,114,475	92MB	0.87 secs
d60	60MB	1,322,746	110MB	1.03 secs
d80	80MB	1,760,962	146MB	1.39 secs
d100	100MB	2,215,851	184MB	1.75 secs
d120	120MB	2,641,700	220MB	2.13 secs
d140	140MB	3,082,257	257MB	2.49 secs
d150	150MB	3,313,863	276MB	2.69 secs
d160	160MB	3,520,162	293MB	2.88 secs
d180	180MB	3,955,286	330MB	3.81 secs
d200	200MB	4,501,816	371MB	4.69 secs
d250	250MB	5,846,239	473MB	6.90 secs
d300	300MB	7,220,387	577MB	9.47 secs
d350	350MB	8,546,961	678MB	11.67 secs
d400	400MB	9,919,241	781MB	13.94 secs
d450	450MB	11,258,413	883MB	16.66 secs
d500	500MB	12,637,069	987MB	19.27 secs

* Total size of the promoted objects, including the 16-byte Sphere overhead

Table 5.1: Load Statistics

node object, an array of references to its children, and an array of references to the data elements (strings, in this case). Each node object also has a reference to its parent node object, which facilitates the B-tree operations. It also renders the object graph more complex by introducing reference cycles.

Table 5.1 gives statistics on the data files used to populate the B-tree. They were generated from articles from the Financial Times newspaper. The files vary in size from 20MB to 500MB and in number of lines from 0.45 to 12.6 million (this is equal to the number of elements added to the B-tree). The promotion size generated by each file is defined as the total size of the objects to be promoted, including the 16-byte Sphere overhead (8-byte object header and 8-byte indirectory entry). Finally, the traversal time included in table 5.1 denotes the time it takes to do a full traversal of the B-tree in main-memory, visiting all its nodes.

All experimental runs involved populating an empty B-tree from one of the data files and then writing all the B-tree objects to a Sphere store through a single promotion operation. The store used was either totally empty or half-populated. Table 5.2 includes some statistics on the generated Sphere stores, giving the number of promoted objects per data file (this ranges from around 0.55 to over 16 million). The average object size was around 65 bytes with and 49 bytes without the Sphere overhead (it varied by a byte or two between different promotion sizes due to variation in the input files). The generated store size and number of partitions for runs against initially empty stores are also given in the table. All runs in this case did not need to create new partitions. During each run, only one partition was allocating at any time. When a partition was used, it had to fill up before another one was picked. All runs with a half-populated store were done against a 930MB store with all of its 929 partitions starting half-full[4].

[4]This was arranged by first populating the store using a special utility that created 50% garbage objects and then garbage collecting it. This store was created once and restored before every run.

Data File	Objects	Store Size*	Partitions*
d20	576,584	40MB	39
d40	1,143,419	78MB	77
d50	1,415,301	96MB	95
d60	1,679,726	113MB	112
d80	2,236,478	151MB	150
d100	2,814,664	189MB	188
d120	3,355,980	225MB	224
d140	3,915,286	263MB	262
d150	4,209,898	282MB	281
d160	4,471,817	300MB	299
d180	5,025,018	337MB	336
d200	5,727,206	379MB	378
d250	7,456,070	482MB	481
d300	9,223,314	588MB	587
d350	10,930,195	690MB	689
d400	12,697,089	795MB	794
d450	14,421,044	899MB	898
d500	16,195,820	1004MB	1003

* When starting with an unpopulated store

Table 5.2: Store Statistics

All times given in this chapter are the average of ten runs, with the worst one removed. The `gettimeofday` function was used to obtain time measurements. Even though more accurate ways to do this exist, this function is accurate enough for the purposes of these experiments, which are fairly coarse-grained and do not require more fine-grained accuracy.

The time measured for a single promotion was the time between creating and committing the history in which the promotion took place. This does not include B-tree population and store startup and shutdown times. In particular, it does not include the flushing of dirty pages that takes place during the latter (this only applies to pages that contain reference count updates and *not* to pages that were written with unlogged writes and have to be transferred to disk before the history can be committed). This is acceptable, as any updated information on such buffers will have been written to the log and could be recreated, if necessary.

All times plotted in this chapter are also compared against a *Base I/O time*. This is the time it takes to write an amount of data equal to the required promotion size(s) to disk using linear writes (the most efficient way to perform them) and the indicated transfer unit size. Timing information for this was obtained using one of the raw partitions used in the experiments.

5.3.3 Summary of Experiments

Table 5.3 summarises the configuration of the experimental runs presented in the following five sections. It includes the data files used per run, the promotion size, Sphere configuration, and which parameters were measured and plotted.

The approach taken, when designing the experiments, was to start with the configuration parameters set to

	§5.3.4	§5.3.5	§5.3.6	§5.3.7	§5.3.8
Load Parameters					
Data File	d20–d200	d20–d200	d200	d20–d200	d50–d500
Promotion Size (MBs)	37–371	37–371	371	37–371	92–987
Promotion Algorithm*	GA/SP	GA	GA	GA	GA
Store Configuration					
Disk Cache (MBs)	6	6	8–48	24	24
Transfer Unit (KBs)	8	8–512	512	512	512
RCUB† Size (KBs)	4	4	4	no/4–64	32
Unpopulated	yes	yes	yes	yes	yes
Half-Populated	yes	yes			
Parameters Measured					
Time	yes	yes	yes	yes	yes
Log Traffic	yes				yes
Allocation Rate					yes

* GA — *Ghosted Allocation*, SP — *Single Pass*
† Reference Count Update Buffer — see section 4.10

Table 5.3: Summary of Experiments and Index to Sections

relatively low values and progressively increase them, focusing on one parameter per section and using what was thought to be the best value for it in subsequent sections. This approach does not fully explore the effect that each parameter has on all of the others. However, attempting to explore all of these effects would have been impractical and this approach highlights the effect that each parameter has on the overall performance of the system.

5.3.4 Comparison to the *Single Pass* Algorithm

In order to demonstrate that the *Ghosted Allocation* algorithm meets its requirement of minimising traffic to the log, it was first compared to an alternative algorithm that was implemented using the same infrastructure (i.e. Sphere and the simple mutator). This algorithm was called *Single Pass*, since it only requires a single pass over the object graph. When an object is visited, it is allocated a PID *and* is also written to the store (unlike the *Ghosted Allocation* algorithm, which performs these two operations in two distinct phases). To ensure fault-tolerance, a log record is written per object allocation, which contains the contents of the object for redo purposes.

When an object is written to the store, other objects that it references might not have already been allocated, therefore their PIDs might not be known. In order to deal with this, all reference fields are initially set to null values, then updated at a later time when all necessary PIDs are available (this generates a second log record to reflect this update). Due to reference cycles, this is not always straightforward to arrange. However, in the case of the B-tree, this was simplified due to its symmetric structure, as illustrated in figure 5.12.

In order to ensure that the comparison between *Ghosted Allocation* and *Single Pass* is as fair as possible, several optimisations were applied to the latter: objects are updated at most once, when all the necessary PIDs are available; when new pages are touched they are not read from the disk, avoiding unnecessary installation reads; the RCUB and unswizzling callbacks used in the *Ghosted Allocation* algorithm were also

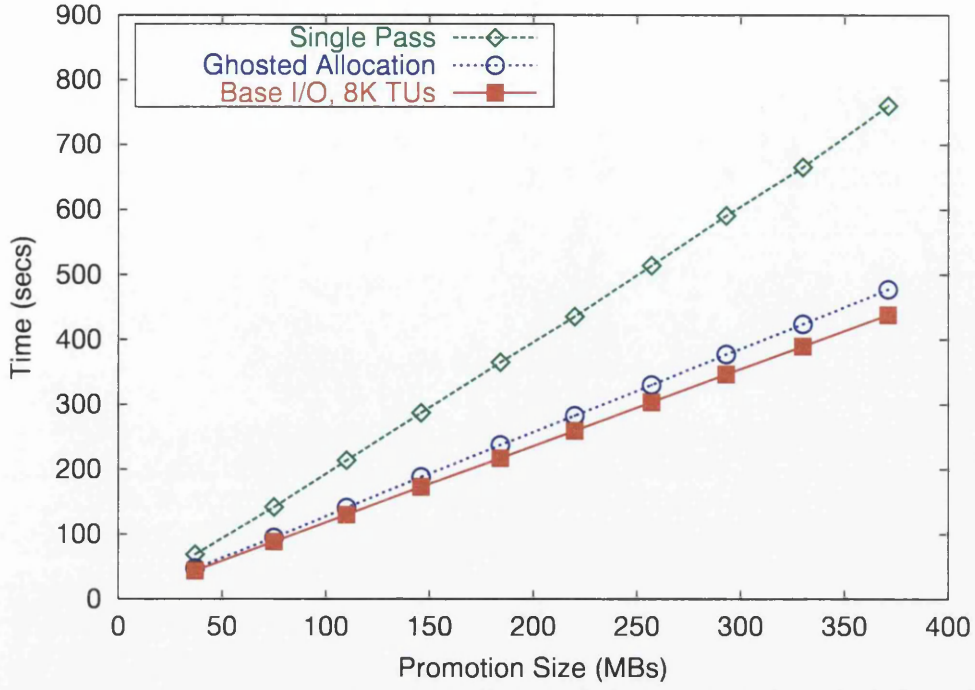


Figure 5.13: *Ghosted Allocation* vs *Single Pass* — Elapsed Time, Unpopulated Stores

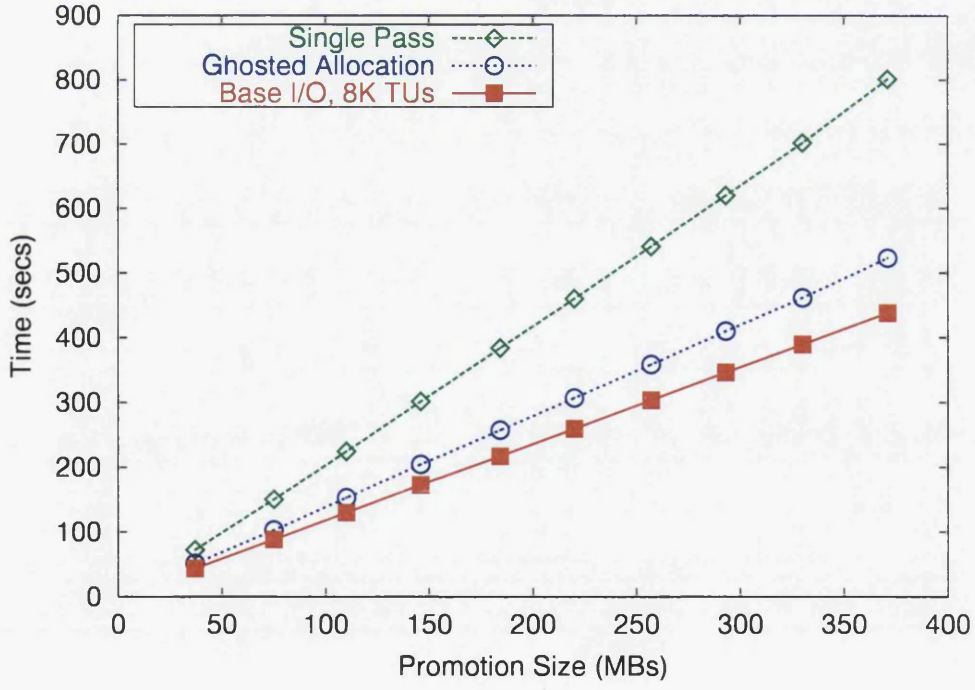


Figure 5.14: *Ghosted Allocation* vs *Single Pass* — Elapsed Time, Half-Populated Stores

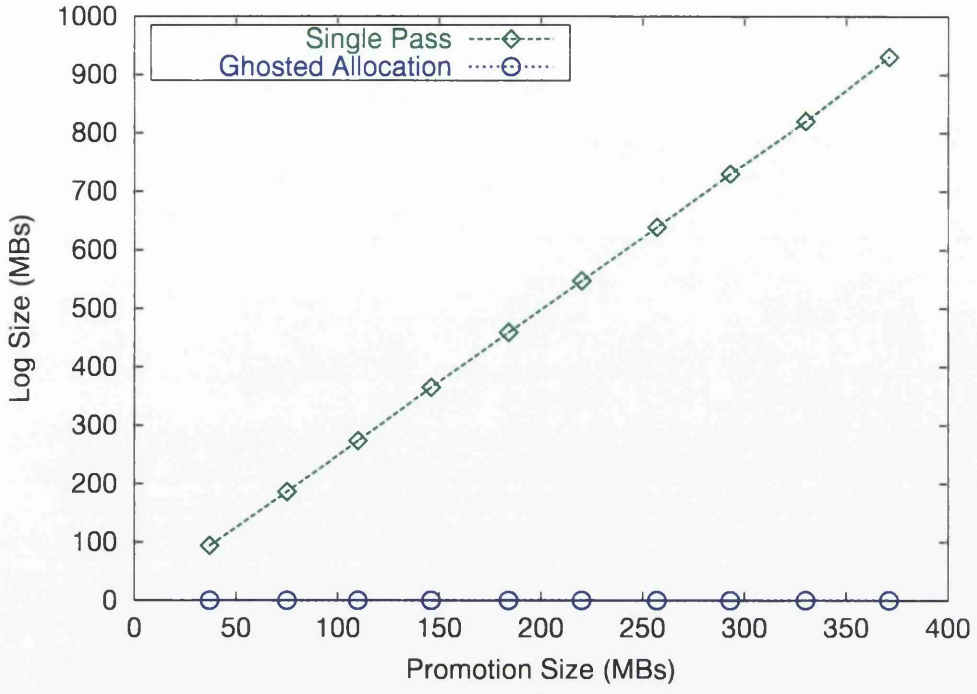


Figure 5.15: *Ghosted Allocation* vs *Single Pass* — Log Traffic, Unpopulated Stores

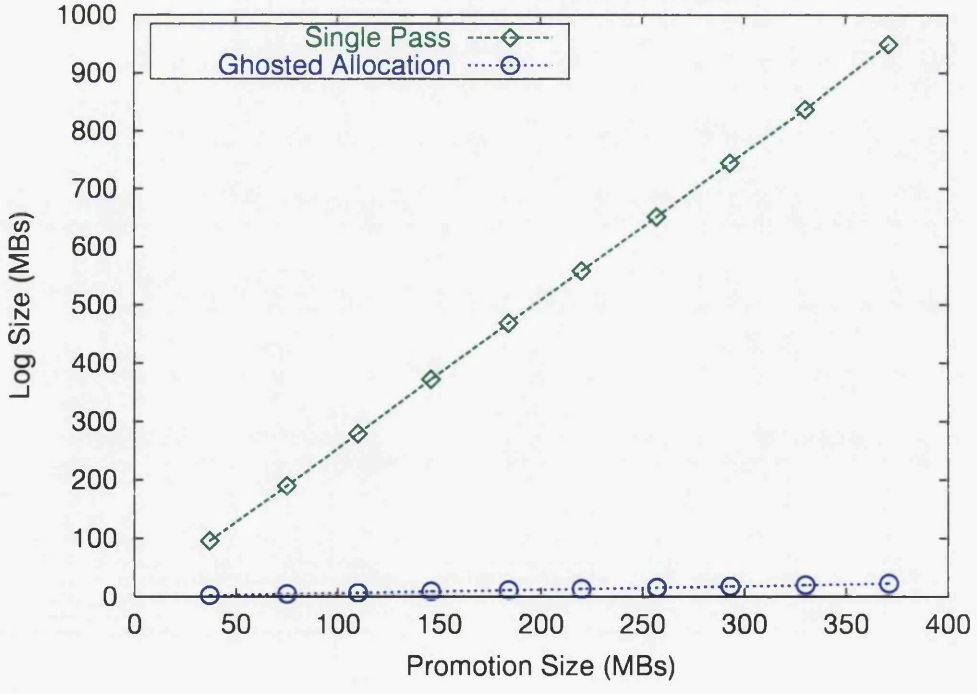


Figure 5.16: *Ghosted Allocation* vs *Single Pass* — Log Traffic, Half-Populated Stores

used.

Figures 5.13 and 5.14 compare the elapsed times for the two algorithms, for unpopulated and half-populated stores respectively, and for a set of promotion sizes between 37MB and 371MB (the elapsed times presented will seem excessive — see section 5.3.5 for explanation and improvements). Even though *Ghosted Allocation* has the overhead of a second object traversal (with a cost shown in table 5.1), it performs about 35% better than *Single Pass* in both cases. Additionally, it is only about 9% slower than the base I/O performance, in the case of the unpopulated stores, and about 19% slower, in the case of the half-populated stores.

Figures 5.15 and 5.16 compare the log traffic of the two algorithms, again for unpopulated and half-populated stores respectively. *Ghosted Allocation* performs considerably better than *Single Pass*, only requiring up to a maximum 0.22MB of log traffic, in the case of the unpopulated stores, or 22.7MB, in the case of the half-populated stores. The two orders of magnitude increase in the latter case was, as expected, due to the logging of parts of half-full pages (↝§5.2.1) and allocations from the indirectory free-list (↝§4.5.3). The corresponding values for *Single Pass* were 930.5MB and 949.5MB respectively.

The results presented in this section show that the *Ghosted Allocation* algorithm does meet its requirement of minimising log traffic by not generating one log record per object allocation. Even though its log traffic does increase, as expected, when half-populated partitions are used, this increase is not excessive if compared to the corresponding promotion size. Also, the overhead of the second object traversal is absorbed by the minimal logging performed. Hence, *Ghosted Allocation* compares favourably to an algorithm that requires a single object traversal. Finally, it is very important to point out that the optimisation that decreases the log traffic employed in *Ghosted Allocation* would not have been straightforward to achieve in the *Single Pass* algorithm. This is also the case for the use of transfer unit sizes larger than the system page size, which is described in the next section.

5.3.5 Sensitivity to Transfer Unit Size

The elapsed times given in the previous section, even though they were acceptable relative to the base I/O time, are poor on an absolute scale. The reason for this can be seen in figure 5.17. Writing the same amount of data using 512K transfer units is almost ten times faster than using 8K transfer units (this is the page size used in the Sphere disk cache). Even though this result was observed on one particular hardware configuration with one particular model of disk drive, it is likely to be similar to other hardware combinations.

The above motivated the use of larger transfer units for the write operations of *Ghosted Allocation*. As described in section 5.2.3, all unlogged writes to the indirectories and object spaces take place linearly (backwards and forward into the partition respectively). It was very straightforward to group several of these in a large buffer and then write this buffer to disk with a single I/O operation. Notice that *only* unlogged writes are performed in this fashion. Any logged writes take place through the disk cache using 8K pages.

This large buffer was introduced as an extension to the Sphere disk cache and it actually bypasses the other buffers used for caching store pages. This allowed the choice of values for its size that are larger than the Sphere page size (8K). This is useful since a buffer size optimal for writing might not necessarily be optimal for reading. Additionally, once the buffer is full or the space that is being written to (i.e. indirectory or object space) is full, the buffer is written to the store synchronously and can be instantly re-used. This avoids unnecessary evictions on the standard disk-cache buffers.

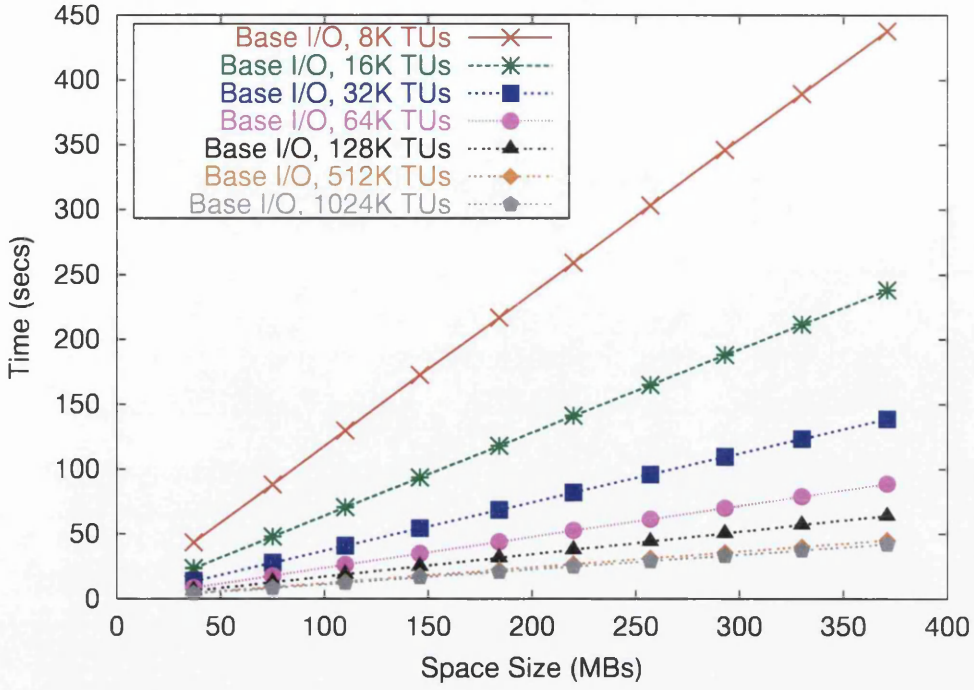


Figure 5.17: Base I/O Performance

The only data that is written using the large buffer, which might be reread during the same promotion operation, are indirectory entries that may be refetched to increment reference counts. Reference count updates only take place during the writing phase and pages containing indirectory entries are written during the allocation phase (⇝§5.2.3). This ensures that when such a page needs to be accessed, it does not reside in the large buffer and is guaranteed to have been written to disk.

Finally, if simultaneous allocations to several partitions are necessary (⇝§5.2.5), then the number of large buffers required to achieve this is equal to the maximum possible number of partitions that are allocating simultaneously (as mentioned in section 5.2.7, this is expected to be low). An alternative approach would be to allocate only a fixed number of large buffers and revert to using single pages when all of these buffers are occupied. This would trade-off performance for lowering the memory requirements.

Figures 5.18 and 5.19 compare the performance of *Ghosted Allocation* when using 8K, 32K, 128K, and 512K transfer units, for unpopulated and half-populated stores respectively. In the figures, hollow data points represent promotion times and filled data points represent base I/O times. An almost five time performance increase is observed when using 512K transfers instead of 8K ones, bringing the absolute elapsed time down to acceptable levels. However, also notice that for all four transfer sizes, the overhead of the *Ghosted Allocation* algorithm over the base I/O performance remains relatively constant in absolute terms. Thus, in percentage terms, it increases from 9% for 8K transfers to 124% for 512K transfers, in the case of unpopulated stores, and it increases from 19% for 8K transfers to 205% for 512K transfers, in the case of half-populated stores. Finally, it is worth noticing that, when using 512K transfers, in the worst case the operation against a half-populated store is 81.2% slower than in the case of an empty store. This is mainly caused by the increase in log traffic and having to write indirectory pages that contain entries allocated from

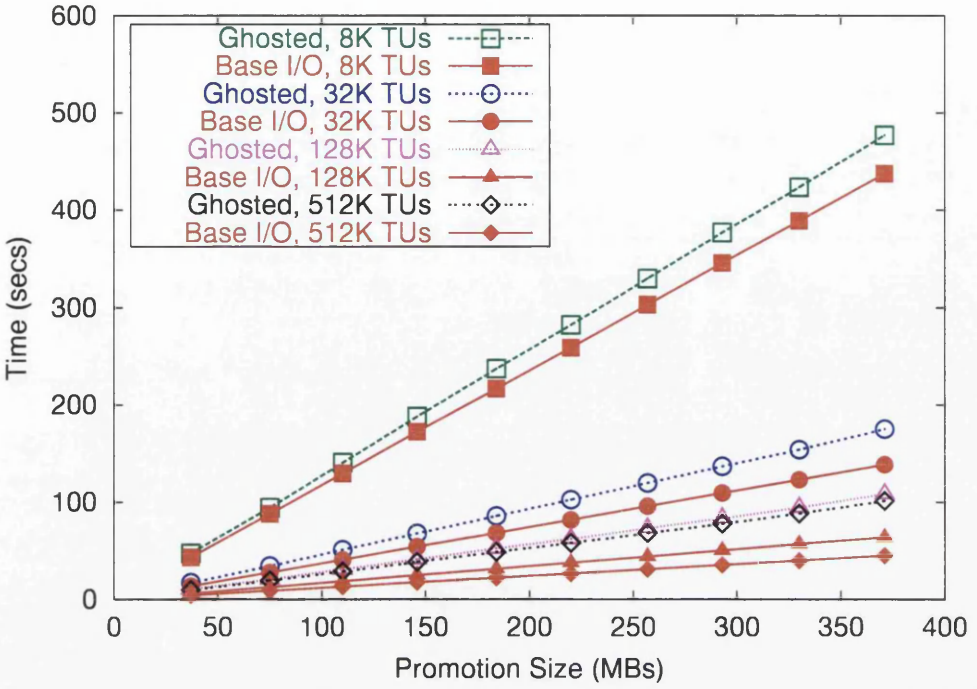


Figure 5.18: Sensitivity to Transfer Unit Size — Elapsed Time, Unpopulated Stores

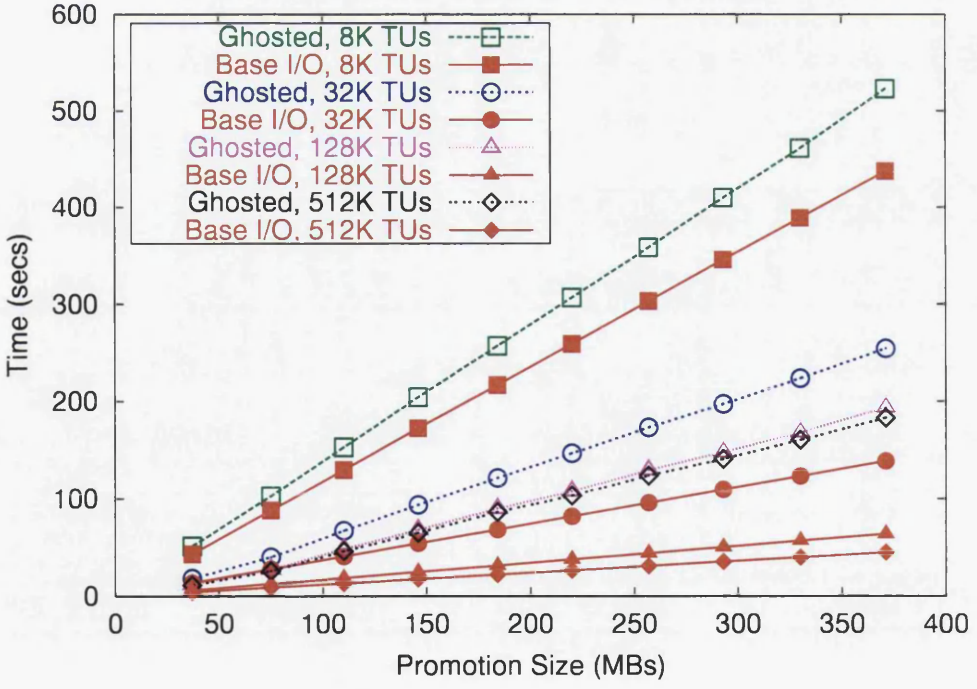


Figure 5.19: Sensitivity to Transfer Unit Size — Elapsed Time, Populated Stores

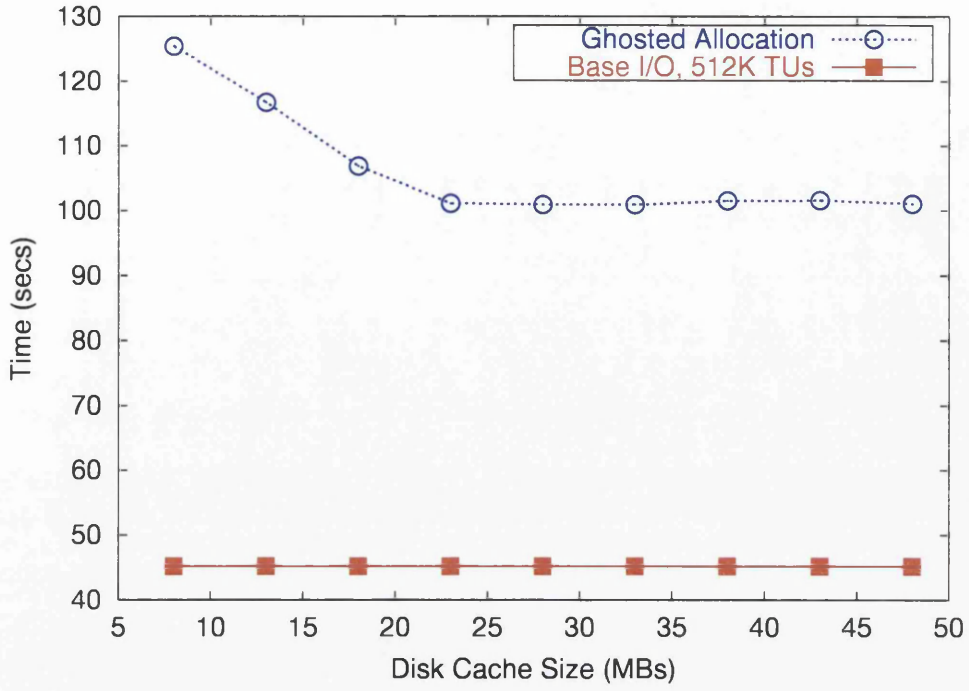


Figure 5.20: Sensitivity to Disk Cache Size — Elapsed Time, Unpopulated Stores, 371MB Promotion

the free-list with 8K transfer units and not take full advantage of the large buffer.

The size of 512K was used for the large buffer for the remaining measurements in this section. However, it must be stressed that not all transfers are done with this transfer size as sometimes the space being written fills up before the buffer and hence only a part of the buffer needs to be written to disk.

5.3.6 Sensitivity to Disk Cache Size

Even though most write operations take place using the large buffer described in the previous section, the disk cache is still used during promotion to fault-in pages that contain parts of the partition table, partition headers, and indirectory entries for reference count updates. Figure 5.20 illustrates the effect that the size of the Sphere disk cache has on the performance of the system when dealing with a 371MB promotion size. An almost 20% performance improvement is observed when the disk-cache size is increased from 8MB to 24MB and then it stays relatively stable for larger values (in fact there is a slight performance degradation for some larger values, the cause of which has not yet been identified). Performance is not improved when more than 24MB were used because all the pages that needed to be faulted-in (the majority being indirectory pages for reference count updates) could fit in 24MB.

Based on this result, a disk cache of 24MB was chosen for the remaining experiments.

5.3.7 Reference Count Overhead

This section explores the overhead of reference count updates and the effect of the RCUB size on the performance of the system. Figure 5.21 illustrates that the performance gain when increasing the RCUB size from

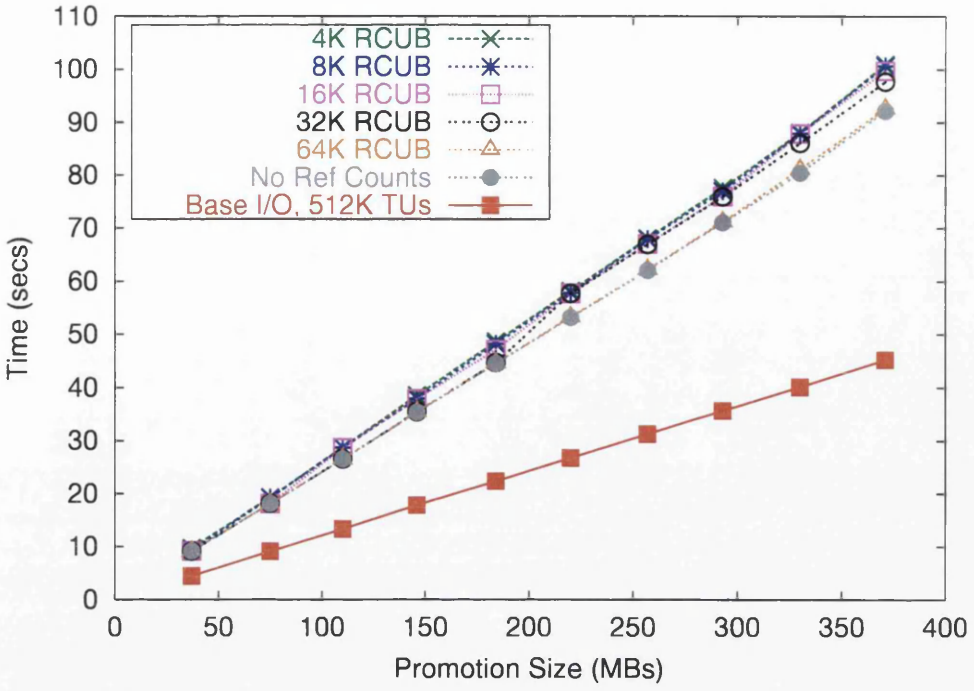


Figure 5.21: Sensitivity to RCUB Size — Elapsed Time, Unpopulated Stores

4K to 64K is up to 8% in the case of the largest promotion size. Additionally, the performance of the system without any reference count updates is also plotted in the same figure. To obtain this the macro that adds new entries to the RCUB was disabled, hence the RCUB buffer was always empty. Notice in the figure that the line for the 64K RCUB is almost identical to the one with no reference count updates. This is because 64K was enough to accommodate all reference count updates of even the largest promotion size used, hence the RCUB was only flushed once.

Based on these results, a 32K RCUB was chosen as the best trade-off between performance and space requirements. For the largest promotion size presented here of 371MB, the overhead of reference count updates when using a 32K RCUB is only around 6% when compared to the case when no reference count updates take place.

5.3.8 Scalability of *Ghosted Allocation*

The final set of measurements taken consisted of an attempt to push the system to its limits and study its scalability with promotion sizes of up to almost 1GB (the largest comprised around 16.2 million objects — see table 5.2)[5]. The system configuration was 512K transfer units, 24MB disk cache, and 32K RCUB. Only unpopulated stores were used in this occasion. The corresponding heap size for the largest promotion size was 864MB.

Figure 5.22 plots the elapsed time of these experiments. The increase in elapsed time is almost linear up to a promotion size of 600MB, imposing an overhead of between 100% and 135% over base I/O. However, it slightly degrades between 600MB and 1GB (i.e. the rate of increase is higher) and reaches an overhead of

[5]Note that this is the largest batch of updates to store in one stabilisation, not the largest POS envisaged.

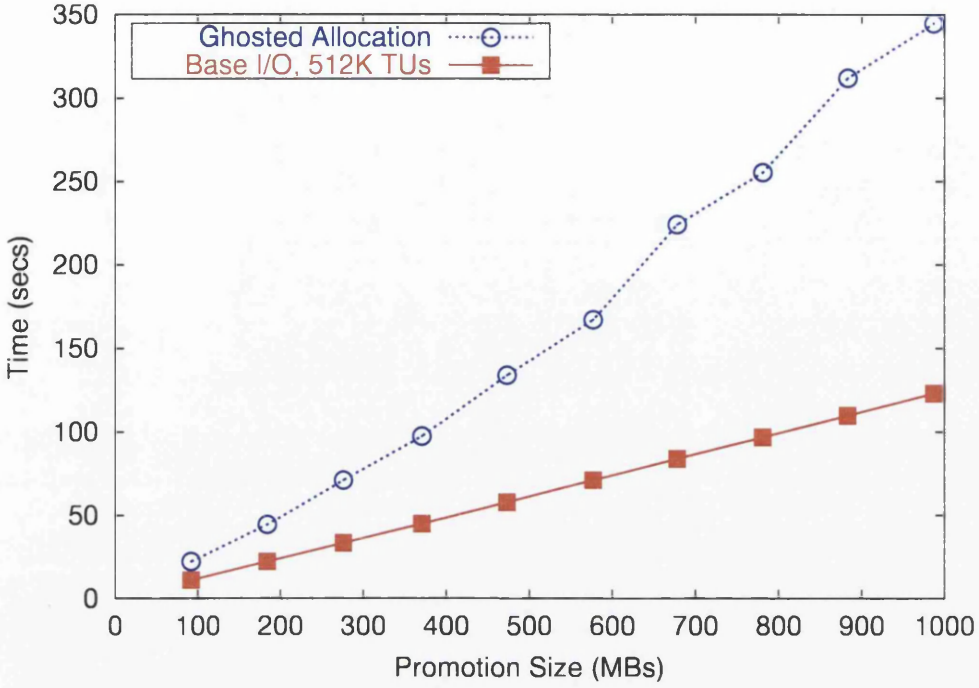


Figure 5.22: Scalability of *Ghosted Allocation* — Elapsed Time, Unpopulated Stores

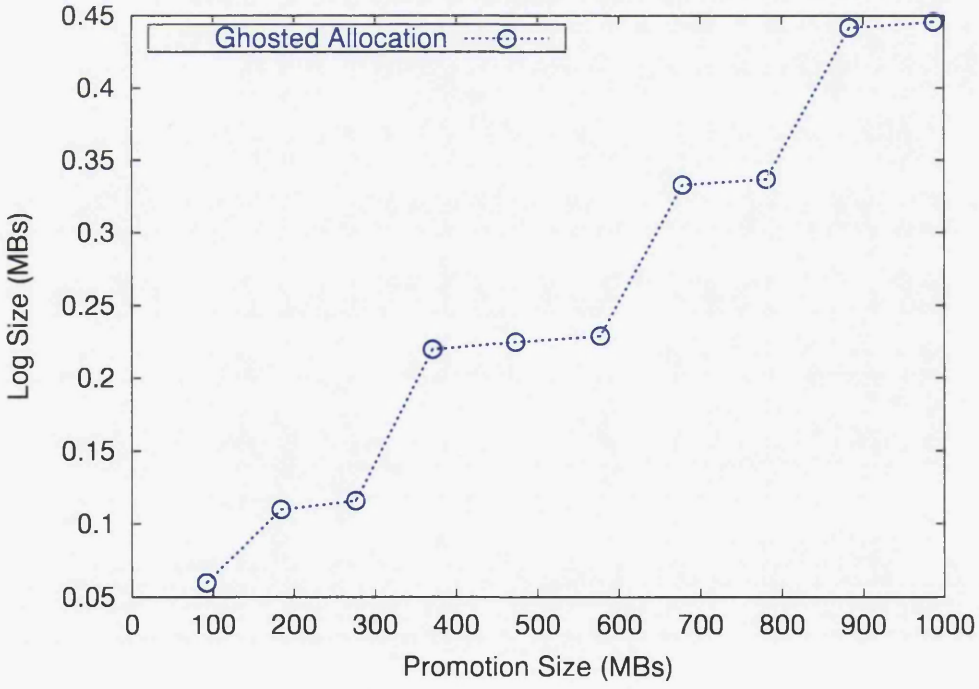


Figure 5.23: Scalability of *Ghosted Allocation* — Log Traffic, Unpopulated Stores

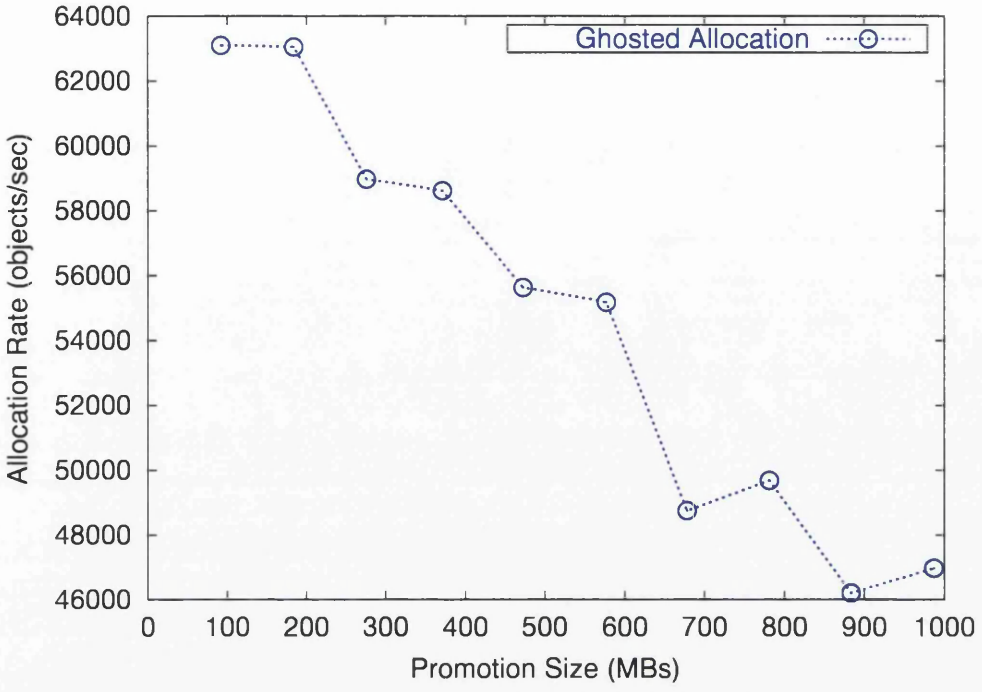


Figure 5.24: Scalability of *Ghosted Allocation* — Object Allocation Rate, Unpopulated Stores

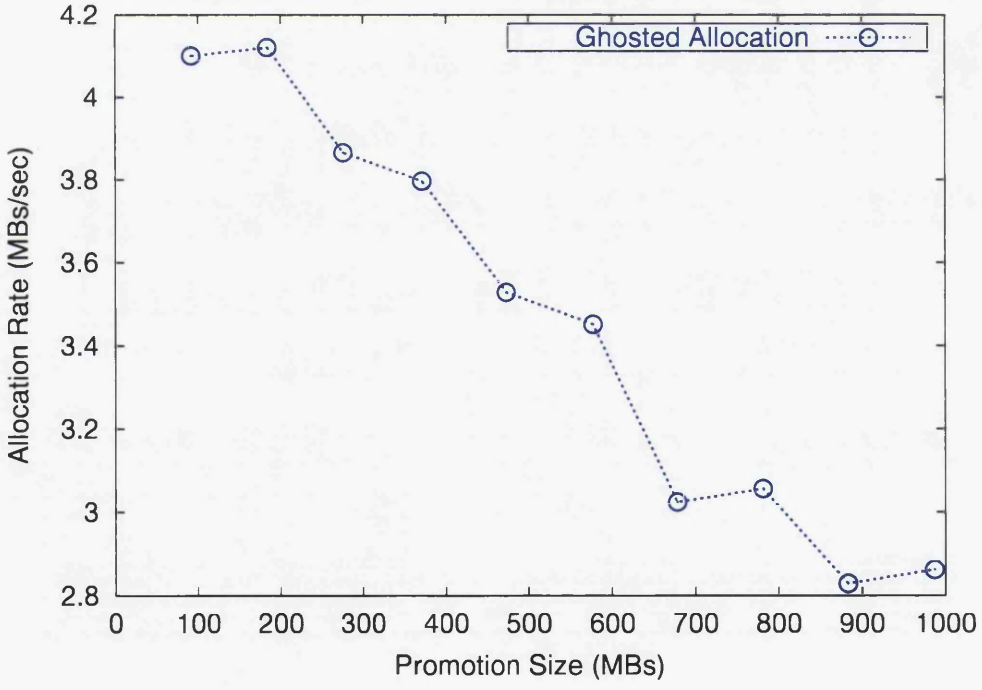


Figure 5.25: Scalability of *Ghosted Allocation* — Space Allocation Rate, Unpopulated Stores

180% over base I/O in the worst case. Also notice the increase after 600MB is not linear. The slight oscillations are caused by RCUB flushes. Still, an elapsed time of 2.8 minutes for a promotion size of 577MB comprising 9.2 million objects is a good result.

The log traffic of these experiments is plotted in figure 5.23. It remains extremely low at under 0.5MB in all cases (0.45MB in the worst case). The three same-sized large increases in log traffic around 300MB, 600MB, and 800MB are caused by RCUB flushes.

The object allocation rate of *Ghosted Allocation*, defined as $(number\ of\ objects)/(elapsed\ time)$, is plotted in figure 5.24. This decreases from just over 63,000 objects/sec down to almost 47,000 objects/sec as the promotion size increases. Also the small degradation in performance after 600MB observed in figure 5.22 is also evident in this figure.

Finally, the space allocation rate of *Ghosted Allocation*, defined as $(promotion\ size)/(elapsed\ time)$, is plotted in figure 5.25. It starts from around 4.1 MB/sec and degrades down to around 2.8 MB/sec for the largest promotion size. The same rate for base I/O was 8.1 MB/sec. Additionally, the shape of the graph is, as expected, very close to the one in figure 5.24.

5.3.9 Summary

This section has shown how the system's performance can be improved by tuning some of its parameters. The largest performance increase was achieved by using larger transfer sizes and some additional improvements were obtained by tuning the size of the disk cache and RCUB. The absolute performance of the algorithm is good, as it can handle a 371MB promotion with 5.7 million objects in 1.63 minutes, with a rate of 58,620 objects/sec and 3.8 MB/sec, only generating 0.22MB of log traffic. The main-memory requirements of the system are low, a 24MB disk cache and a 32K RCUB, that is only about 6.5% of the promotion size. They can be decreased further, if necessary, by trading off performance for a smaller memory footprint. Notice also that the main-memory overhead in the system is the main-memory heap containing the objects and not the store data structures.

Admittedly, the performance of the system degrades when the promotion size is stretched to almost 1GB. This could have been improved upon, if the parameters of the system were tuned for these large sizes (this was only done for promotion sizes up to 371MB). Single promotions that large are rare in practice, therefore the system was tuned for smaller sizes, as a trade-off between performance and memory requirements. Further tuning could improve performance for other workloads. It is currently being considered to allow the users to set several of the store parameters mentioned in this section by editing the Sphere configuration file (⇝§C).

The 124% overhead of the algorithm over the base I/O time can be decreased further by increasing the use of asynchronous disk transfers and performing more tuning and code profiling. Note however that 18% of they overhead is the two in-memory object traversals (see table 5.2).

5.4 Related Work

In this section, a few systems whose architecture is similar to the one assumed in this chapter (↝§5.1) are overviewed. Sections 5.4.1 and 5.4.2 visit respectively the Napier88 and Tycoon orthogonally persistent systems. Section 5.4.3 describes the promotion algorithm of PJama Classic, the system the *Ghosted Allocation* algorithm was designed to improve upon. Finally, section 5.4.4 discusses a system with slightly different architecture: the Persistent Modula-3 system whose faulting mechanism is page-based. The four systems are presented chronologically. They are summarised and compared to the *Ghosted Allocation* algorithm in section 5.4.5.

5.4.1 Napier88

Napier88 [MCC+99] is an orthogonally persistent programming language and environment, developed at St Andrews University, Scotland. It was used as a vehicle for research into efficient implementation techniques to support orthogonal persistence [Bro89] as well advanced programming language features, like parametric polymorphism and high-order procedures and functions [MCC+99, CCM99]. For its storage requirements it relies on a custom-built yet generic POS [BM92, BR91, BMM+99]. Interestingly enough, in Napier88 the single-space persistent heap, which is used to cache persistent objects and accommodate newly-allocated ones, is considered part of the POS rather than a separate component [Bro89]. Therefore, the heap and the storage mechanism are very tightly-coupled.

Newly-allocated objects are not given a PID upon allocation, in case they are garbage collected before a stabilisation operation. This is often the case, as the "death-rate" of objects in Napier88 is very high; around 75% of objects do not survive the next 100 allocations [PC96]. However, all allocated objects are considered persistent and may be written to the POS (unlike the PJama system [PAJ99]). Additionally, all active threads are persistent. It follows that, during a stabilisation, all objects that are not garbage (i.e. those that are reachable from the runtime stacks) must be made persistent.

Given the above assumptions, the stabilisation operation of Napier88 is relatively simple. First, a heap garbage collection is performed to remove all garbage objects. Then, the heap is linearly scanned and all newly-allocated objects are allocated PIDs and space in the POS (notice that no complex object-scanning phase is necessary to find them, as all objects in the heap are either persistent or must become persistent). As new objects are always appended linearly to the Napier88 POS and the addressing scheme it uses is physical (i.e. the PID can be deduced from the location of the object in the POS), the allocation operation can be done entirely in memory with no disk writes being necessary. After all objects in the heap have been allocated a PID, they are unswizzled and written to the store. No eager reswizzling takes place but the normal operation of the Napier88 interpreter will lazily swizzle them when they are next dereferenced.

5.4.2 Tycoon

Tycoon [MSS99] is an orthogonally persistent environment developed at Hamburg University in Germany. It includes the custom-designed TL language for database application programming [Mat99, MMS94]. The Tycoon architecture is close to the one assumed in this chapter: the POS is separated from the single-space persistent heap and is accessed through a well-defined interface, called the *Tycoon Store Protocol* (TSP) [MSS99, MMS99]. This has allowed the Tycoon system to use a variety of POSs with only the TSP layer needing to be adapted accordingly. Implementations of TSP for ObjectStore [LLOW91], the Napier store

[BM92], and custom-built POSs have been reported [MMS99].

Unlike systems such as Napier88 [Bro89] and PJama Classic [DA97], the Tycoon system does not perform any pointer-swizzling; i.e. all reference fields in objects in memory always contain PIDs rather than memory references to other objects. Additionally, when an object is allocated in the persistent heap, it is always allocated a PID and space in the POS. Given the previous two points, stabilisation in the Tycoon system is very straightforward. All modified objects in the persistent heap are copied to the POS, with no further transformations being necessary. Promotion, as described in this chapter, is not a part of the stabilisation operation as all objects have already been allocated to the POS upon creation. The costs incurred to avoid promotion may re-appear as unnecessary PID and space allocations and as higher main-memory pointer dereferencing costs. The increased allocations in the POS also increase the resources needed for disk garbage collection.

5.4.3 PJama Classic

The PJama system (↝§2) is an orthogonally persistent environment for the Java programming language [GJS96]. Here is described the promotion operation of its prototype implementation, referred to as PJama Classic [AJ00].

PJama Classic is been built on top of a custom-built physically-addressed POS that depends on RVM [MS94] for its recovery facilities. The persistent heap of PJama Classic is split into two components: the transient heap, where new objects are allocated, and the object cache, where persistent objects reside [DA97]. The invariant that is maintained is that all objects in the object cache have been allocated a PID and all objects in the transient heap have not. It was originally intended for the persistent heap to be decoupled from the POS, however this was not eventually achieved, as described below.

In the PJama system some references can be marked as transient by a programmer so that they do not cause their referenced objects to be promoted to the POS (↝§2.3.4). A stabilisation initiates a recursive object-graph traversal operation, which starts from all modified reference fields in persistent objects, to discover the objects that should be written to the POS. Any objects that are visited during this traversal are copied from the transient heap to a number of contiguous areas in the object cache, called *promotion regions*, where they are unswizzled and given a PID [DA97]. In a similar manner to the Napier88 system, all PID allocations are done entirely in memory and no disk accesses are necessary. When all required objects have been copied to the object cache, the promotion regions are written to disk, extending the POS using the technique described in section 5.2.1. Finally, the newly-copied objects in the object cache are reswizzled before normal operation is resumed. This reswizzling is necessary since it also re-installs the values of transient fields that were set to defaults when written to disk [PAJ99].

5.4.4 Persistent Modula-3

The Persistent Modula-3 (PM3) system [HC99b, HC99a], developed at Purdue University, USA, is an orthogonally persistent version of the Modula-3 programming language [CDG+89]. It differs from the other systems presented in this section as its heap implementation has ambiguous garbage collection roots [BW88, Jon96]. The storage system used by PM3 is the SHORE store [CDF+94], but according to the PM3 developers any general-purpose POS can be adopted [Hos99].

To implement persistence by reachability, PM3 uses an adapted version of Bartlett's mostly-copying garbage collector [Bar88, Bar89], described in detail by Hosking and Chen [HC99a]. It allows ambiguous garbage

	Napier88 §5.4.1	Tycoon §5.4.2	PJama §5.4.3	PM3 §5.4.4	Ghosted §5.2
Persistent Heap					
Transient Objects	yes*		yes	yes	yes
Swizzling	yes		yes	yes	yes
Space Overhead			double		
Persistent Store					
Decoupled		yes		yes	yes
Generic		yes		yes	
Transfers to POS	objects	objects	objects	pages	objects
Addressing	physical		physical		logical
Disk GC	off-line	none†	off-line	none†	concurrent

* All objects are persistent, however they are not immediately allocated a PID
† The author is not aware that a disk garbage collector exists for these two systems

Table 5.4: Comparison between the Related Work and *Ghosted Allocation*

collection roots but assumes that the root of persistence and all reference fields in objects can be determined unambiguously. It splits the heap into pages and during garbage collection "copies" live objects by flipping the status of the page where they reside from "from-space" to "to-space". Moving objects between pages is also allowed, provided that they are only reachable non-ambiguously (i.e. they are only reachable from other objects rather than the ambiguous roots).

The transfers between the heap and the store are done in entire pages rather than individual objects. Each page is stored as a single SHORE object and is given a unique PID. The object PIDs in each page are calculated from the page PID plus their location within the page. When a page is faulted-in, the reference fields in it are swizzled using the pointer-swizzling at page-fault time technique [WK92]. The operating system's memory-protection facilities are used to keep track of which pages are updated during program execution.

All pages in the heap are marked either as transient or persistent. A stabilisation operation discovers modified persistent pages and updates their store image. Additionally, any transient pages that are reachable from modified persistent pages are also written to the store and their status is changed to persistent. During this operation, care is taken to recluster objects wherever possible to minimise the number of new pages transferred to disk. When a page is written to the store, reference fields in it are unswizzled with a mechanism similar to the one described in section 5.2.4 [Hos99].

5.4.5 Summary

Table 5.4 summarises the promotion mechanisms of the systems described in this section and compares them with the *Ghosted Allocation* algorithm. Of the five, Tycoon's is the most straightforward, aided by the simplicity of its architecture. Additionally, the TSP layer (⇝§5.4.2) keeps the POS and persistent heap decoupled and allows various POSs to be adopted. However, both the lack of swizzling and the allocation of PIDs at object-allocation time impose runtime overheads and scalability problems.

In both PJama Classic and Napier88 systems, all allocations in the POS take place entirely in memory. This is made easier by the physical addressing scheme adopted in both systems and by the fact that new objects

are always allocated linearly at the end of the store file. However, both systems depend on custom-built storage systems with tight-coupling to the persistent heap. The PJama Classic promotion mechanism is more complicated than that of Napier88 because it has to distinguish between transient and persistent references and objects, whereas Napier88 assumes everything is persistent. However, the evacuation of objects from the transient heap to the object cache in PJama Classic imposes equal space requirements in both spaces, essentially doubling the memory required. This has caused problems with applications that require large bulk object-loading operations.

The PM3 system has been built around a particular implementation of the Modula-3 language and has been based on its assumptions. This has essentially encouraged the page-wise, instead of object-wise, transfers to the POS. It is the only system discussed here that is not written in C. Instead it is written in Modula-3 and accesses the POS through a Modula-3-to-C interface, which presumably also keeps the two well-abstracted. Even though impressive performance results have been reported using PM3 [HC99b, HC99a], its scalability has not yet been studied, nor has the effect that the page-wise transfers and conservatism have on its performance.

The *Ghosted Allocation* algorithm compares well to the above systems. Even though it only applies to a POS architecture similar to Sphere, it does not require tight-coupling between the POS and the persistent heap and the requirements that it imposes on the heap are easily implemented. Finally, as noted in table 5.4, it is the only one that supports logical PIDs and concurrent disk garbage collection.

5.5 Summary

This chapter described *Ghosted Allocation*, the efficient and scalable bulk object-loading mechanism implemented in Sphere. The motivation behind this work was first given (⇝§5.1). Then, the *Ghosted Allocation* algorithm was described in detail (⇝§5.2) and, based on experimental results it was shown that it is efficient and scalable (⇝§5.3). Finally, the related work was presented (⇝§5.4).

The next chapter deals with the disk garbage collection framework implemented in Sphere.

Wipe them out, all of them.
*— **Darth Sidious**, Dark Lord of the Sith*

Chapter 6

Garbage Collection in Sphere

This chapter discusses disk garbage collection issues in Sphere. Section 6.1 discusses why garbage collection is necessary in the context of POSs and how it compares to the traditional techniques. Section 6.2 summarises the garbage collection framework of Sphere. Section 6.3 gives a detailed description of the Sphere compacting garbage collector and section 6.4 evaluates it by measuring several aspects of it when running against two benchmarks. Section 6.5 discusses the interaction between the disk garbage collector and the mutator, section 6.6 introduces the off-line Sphere global garbage collector, and section 6.7 describes how the compacting garbage collector was used as the basis of Sphere's evolution facilities for PJama_1. Future work is given in section 6.8 and related work in section 6.9. Section 6.10 concludes the chapter.

6.1 Persistence and Garbage Collection

This section gives a brief overview of garbage collection in a persistent context. Section 6.1.1 gives the reasons why it is necessary and section 6.1.2 compares it with traditional main-memory and distributed garbage collection.

6.1.1 The Case for Persistent Garbage Collection

Traditional main-memory garbage collection, i.e. the *"automatic reclamation of unused storage"* [Jon96], has existed since the 1960s but has only recently become widely accepted and used, thanks to the success of the Java language [GJS96]. Its main benefit is that it eliminates the two traditional programming errors that can occur when explicit memory management is used: *memory leaks*, caused by unused memory not being properly de-allocated, and *dangling pointers*, caused by used memory being wrongfully de-allocated.

As a POS has the notion of a persistent reference (e.g. through PIDs), reference updates in persistent objects can render other objects unreachable. Such objects cannot be accessed any more (because a mutator can only discover PIDs in objects it can access transitively from the persistent root[1]), hence they need to be

[1] In other POSs this is not strictly the case since iterations over the store, e.g. to execute queries, might discover unreachable objects. In Sphere schema evolution is such an operation. However, it has been instrumented to only also perform a global marking phase (⇝§6.6) and only discover live objects.

reclaimed so that the space they take up can be re-used. If an explicit de-allocation policy is adopted (this is the case for many POSs, such as ObjectStore [LLOW91], Texas [SKW92], etc.), programming errors can cause the two problems mentioned above, jeopardising the referential integrity of the store. However, in the case of a POS, such errors have more severe consequences, as faults persist and can permanently corrupt the store and render it unusable. This can be particularly pernicious when one program corrupts the store's contents and this is not discovered until much later. By this time it may be impossible to track down the program that caused the corruption and many other programs may have added valuable data to the store. The adoption of a garbage collector in a POS has the potential to deal with the above problem. The case for referential integrity is further reinforced by the following quote.

> *"We build mission critical military systems. Availability is so important that these systems often include dual-redundant hardware and software. However, even dual redundancy will not automatically protect an application from a referential integrity fault. This is because when the main application attempts to access an object by following a 'dangling' pointer (that is, the pointer is not null but points to a deleted object), it will raise an exception and probably fail. Since the 'hot spare' application is a mirror image with the exact same data, it will also fail for the same reason. Referential integrity is therefore essential, since the alternative would be to put dangling pointer checks or exception blocks everywhere an object is accessed."*
>
> — Michael P. Card, Lockheed Martin

Another issue is that, in the case of orthogonally persistent programming languages, it is desirable that the same model of space management is adopted both in the store and in main-memory, to allow the programmer to consistently and orthogonally manage objects using the same model irrespective of where the object resides [AJ00]. As Sphere was originally designed to support Java objects, it was only natural to adopt a garbage collector for its free-space management, which compliments the main-memory garbage collector of Java. Some similar systems have taken a similar approach (e.g. GemStone/J). Still, there are some persistence solutions for Java that only support explicit persistent object de-allocation (e.g. the system from POET Software [POEwww]).

6.1.2 Comparisons

When comparing a traditional main-memory garbage collector to a persistent one, there are six important points why the latter is more difficult to design and implement. These are enumerated below (some of them are also discussed by Amsaleg *et al.* [AFG95]).

❶ **Size** — POSs are envisaged to be several orders of magnitude larger than main-memory heaps and their expected size is growing steadily. Hence, techniques that perform well in the latter context, do not necessarily apply in a persistent context, e.g. generational techniques [LH83, Ung84] do not apply as the oldest generation will be prohibitively large.

❷ **Longevity** — POSs are built to be long-lived, as opposed to memory heaps that most of the time support relatively short-lived applications (there are of course exceptions to this, e.g. Web and e-mail servers). Because of this, problems that might become more severe over time, e.g. fragmentation, need to be dealt with more carefully.

❸ **Access Time** — The access time of disks is typically orders of magnitude slower than that of main-memory (e.g. the average seek time of the Fujitsu MAB3091 hard disk is 7.5ms for reads and 8.5ms for writes [Fujwww], the access time of EDO memory for PCs is typically 60ns). Hence techniques

that apply to main-memory garbage collection, e.g. include the mark bits on the object header, have to revised in the former case to reach an efficient solution.

❹ **Fault-Tolerance** — One important requirement for a persistent garbage collector is to be fault-tolerant and never leave the POS in an inconsistent state. Some traditional garbage collection approaches, e.g. in-place compaction, complicate this requirement considerably as they overwrite data that might be needed during recovery after a failure.

❺ **Caching** — Typically, there is an object cache between a POS and the mutator where persistent objects are cached. This replication introduces subtle problems, such as the complication discussed in section 6.5.

❻ **Transactions** — A transaction rollback can invalidate the fundamental assumption of garbage collection that *"when an object becomes garbage, it stays garbage"* [AFG95]. This can complicate the implementation of a disk garbage collector and requires some interaction between the garbage collector and the transaction manager of the system.

As discussed in section 3.4.3, one approach to disk garbage collection is to organise the stores into partitions and process one such partition at a time [Bis77, YNY94, AFG95]. One can observe that this architecture is similar to the one assumed for distributed garbage collection [PS95], with partitions being similar to nodes, cross-partition references having to be maintained, etc. However, there are some important differences between the two areas.

❶ **No failures** — Distributed garbage collection has to deal with failures and nodes not being available for periods of time. This is not so in the case of a POS (at least under the assumptions of Sphere), as all partitions are assumed to be available.

❷ **No out-of-order message delivery** — As nodes communicate with each other over a network, the messages they send cannot be assumed to be delivered in the same order (or at all). This does not happen in a persistent context (unless the store is distributed over different machines).

❸ **Data structure maintenance** — As distributed garbage collection algorithms are usually targetted for main memory, using complex auxiliary data structures is relatively straightforward (even though in some cases small parts of these data structures need to be persistent to survive failures). However, maintaining complex data structures on disk is more troublesome as they *(i)* are more expensive to access, *(ii)* have to be well planned so appropriate disk space is reserved, and *(iii)* caching facilities for them usually have to be implemented.

Currently, the storage requirements for POSs are in the order of several gigabytes. However, in the future this figure will undoubtedly shift to the order of terabytes. This in conjunction with other factors; e.g. availability through replication, will increase the distribution of the data, and hence introduce some of the complications described above to the area of persistent garbage collection. In fact, there is evidence that this is already the case [ML97].

6.2 The Garbage Collection Framework of Sphere

One of the fundamental requirements imposed on the design and implementation of Sphere was to support incremental disk garbage collection (⇝§3.2). The features of Sphere that were introduced to achieve this are summarised below.

- **Partitions (↝§3.4.3)** — Partitions are object containers. They are self-contained in the respect that they include complete information on which objects are reachable from other partitions and/or the persistent root (in Sphere, this is achieved by the management of cross-partition reference counts, ↝§3.7.2). This allows them to be garbage collected independently and they can be considered as garbage collection units. This bounds the time that an invocation of the garbage collector will last, which otherwise would be proportional to the size of the entire store. Additionally, it allows the POS to optimise the scheduling of when and which partition will be garbage collected in order to increase throughput and hence the effectiveness of garbage collection [CWZ94, CKWZ96]. The use of partitions for the purposes of disk garbage collection is widely-known and adopted [YNY94, AFG95, ML97].

- **Logical Addressing Model (↝§3.7.1)** — In Sphere, the representation of PIDs used to identify persistent objects does not depend on the location of the objects or the partitions that contain them. In this way, an object can be moved inside a partition and the partition can be moved inside the store without the object's PID changing. This is essential if compacting garbage collection algorithms are to be considered. Alternatively, if a physical addressing scheme was adopted, it would require PID-containing fields in objects to be updated to reflect the changes in the position of objects. This requirement is simply impractical in very large POSs.

- **Multiple Garbage Collection Mechanisms (↝§3.5.1)** — The introduction of the abstraction of partition regimes allows Sphere to adopt different free-space management for different partitions, optimised for the kind of objects they contain. As a garbage collection algorithm is tightly coupled with the free-space management scheme adopted by the space (e.g. POS, heap) over which it operates, the introduction of partition regimes also allows different garbage collection techniques to be introduced, optimised for the objects over which they will operate (e.g. compaction for small objects, in-place de-allocation for very large objects).

- **Thread Synchronisation (↝§4.9.2)** — As described in section 4.9.2, when a thread needs to access an object (to fetch it, update it, etc.), it has to perform a look-up on the PLC to determine the location of the partition that contains it. Further, the PLC entry that corresponds to the partition is kept pinned throughout the operation (↝§4.9.2). This provides an effective way of determining at any time whether a partition is being accessed or not. Additionally, the required PLC look-up conveniently provides a single point at which to block threads from accessing a partition, when required (↝§6.3.1.1).

- **Synchronisation with Updates (↝§4.8.5)** — The mutual exclusion of accesses to a partition by writer threads (which perform updates on objects, reference count updates on indirectories, or promotion) and the garbage collector ensures that the latter always accesses the partition in a consistent state. This avoids any further interaction between the GC and the writer threads. Especially, it avoids the typical synchronisation necessary to ensure completeness and correctness in the case of incremental garbage collectors [Jon96, Wil92]. However, a complication with this approach does arise because of the caching of objects at the mutator. This issue is discussed in section 6.5.

6.3 The Sphere Compacting Garbage Collector

Even though the Sphere framework can support different free-space management schemes co-existing in the same store (↝§3.5.1), only one has been implemented so far (↝§4.13.2). It relies on compaction to reclaim free space (↝§3.8.1) and this ensures that no fragmentation-related problems will occur in long-lived

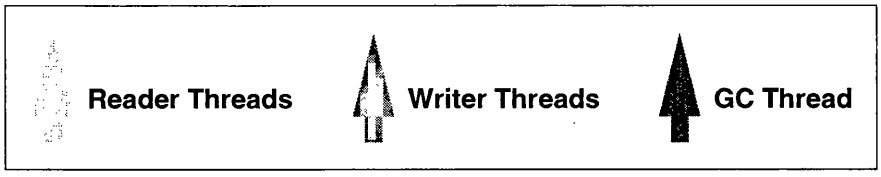


Figure 6.1: Compacting Garbage Collection — Thread Synchronisation — Legend

partitions. For the remainder of this chapter, the Sphere compacting garbage collector will be referred to as the *Compacting GC* or simply the *GC*. As was the case in the promotion experiments (↝§5), the implementation of the GC was totally outside the Sphere core, apart of course from the partition locking (↝§4.8.5) and thread synchronisation (↝§4.9.2) code that are generic.

Section 6.3.1 gives an overview of the algorithm, concentrating on thread synchronisation (↝§6.3.1.1) and fault-tolerance issues (↝§6.3.1.2). Section 6.3.2 describes the main-memory data structures used by the GC and section 6.3.3 the analysis performed to determine whether the GC should complete its operation or abort prematurely. Section 6.3.4 gives a detailed description of the operation of the GC using a concrete example. Section 6.3.5 discusses why retaining the allocation order of objects is essential when processing a partition and section 6.3.6 summarises the section.

6.3.1 Overview

The adoption of a garbage collection algorithm that compacts a partition in-place would not be straightforward to implement as

- synchronisation would be needed to allow reader threads to access objects in the partition that might be in the process of being moved and
- aborting the garbage collection operation fault-tolerantly would be difficult, as objects can be overwritten by others during compaction and their contents might have to be replicated in the log to ensure that they can be recovered.

In order to deal with these two issues, a two-space scheme [Che70, Bak78, Jon96, Wil92] was chosen instead. In particular, an adaptation of the *Replication-Based Garbage Collection* algorithm was implemented [NOPH93], which differs from the standard two-space schemes in that the mutator is concurrently accessing from-space instead of to-space. For this to happen safely, the GC does not update from-space at all.

The operation of the GC is described in section 6.3.4. However, the following two sections deal with the thread synchronisation mechanism (↝§6.3.1.1) and the fault-tolerance issues of the algorithm (↝§6.3.1.2).

6.3.1.1 Thread Synchronisation

The GC moves the partition it operates on to a different location inside the store. It is therefore necessary to ensure that any threads that are accessing that partition are redirected to its new location upon completion of the GC operation. The mechanism for this is described in this section. Figure 6.1 provides the legend for the figures.

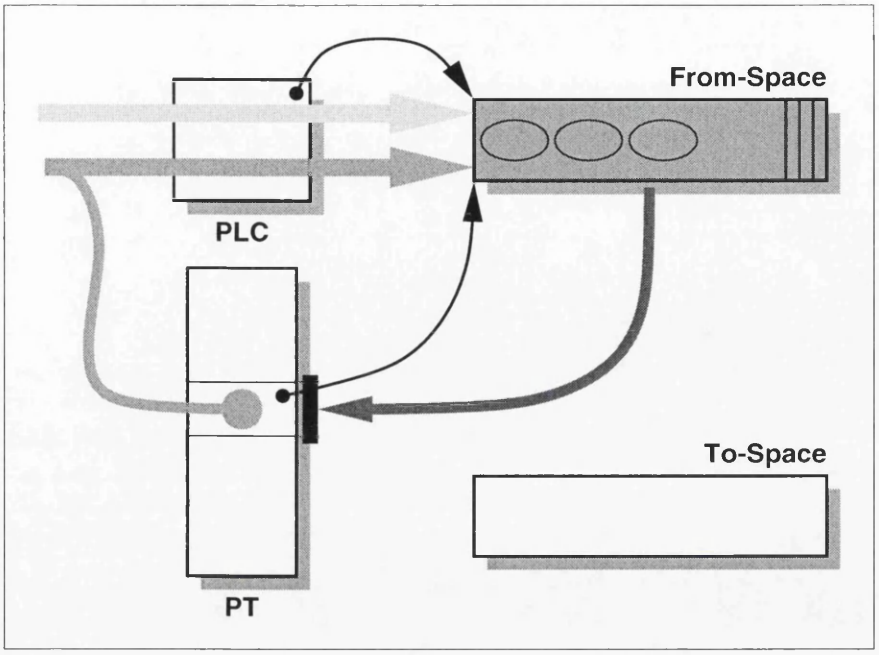


Figure 6.2: Compacting Garbage Collection — Thread Synchronisation — Start

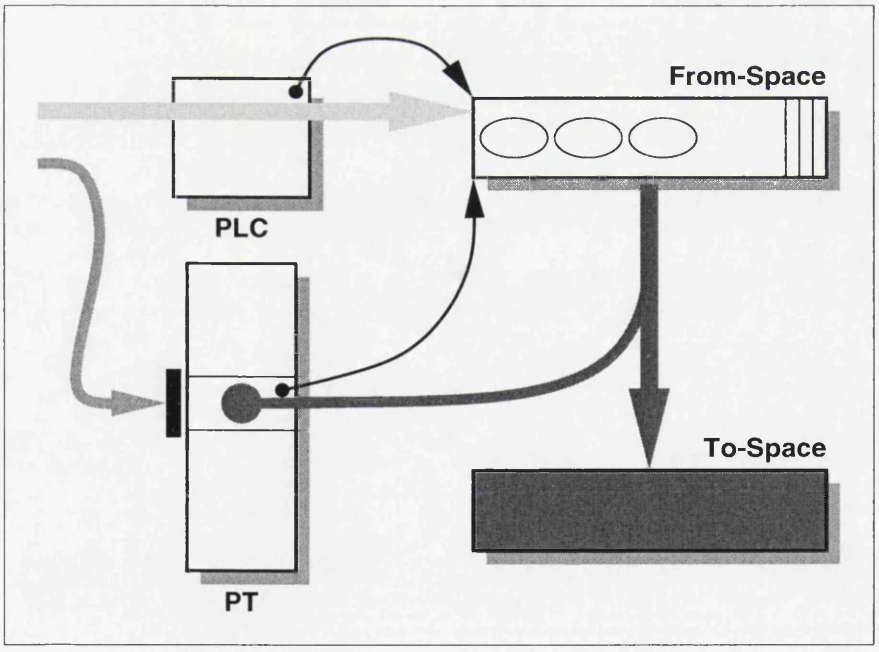


Figure 6.3: Compacting Garbage Collection — Thread Synchronisation — Blocked Writers

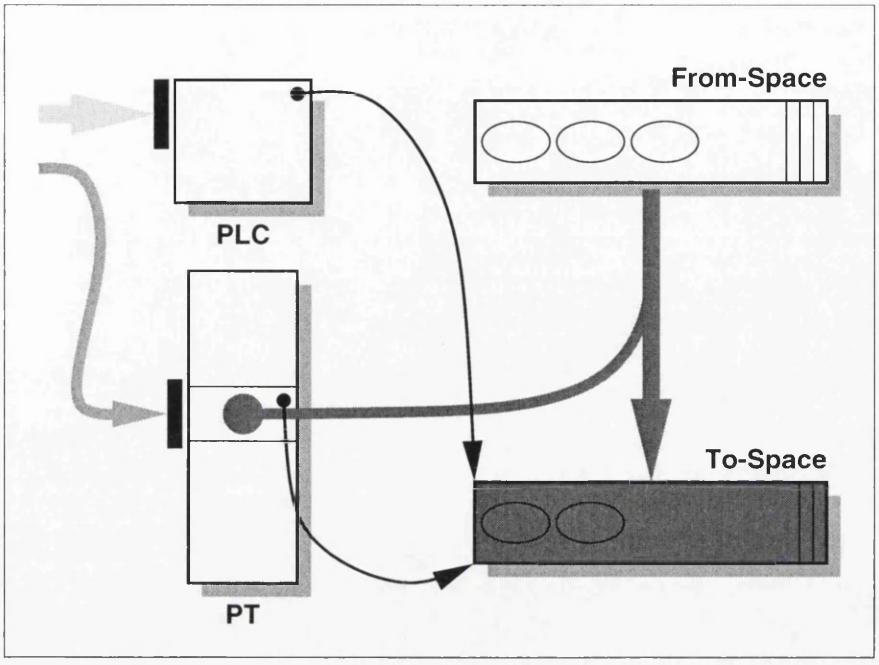


Figure 6.4: Compacting Garbage Collection — Thread Synchronisation — Blocked Readers

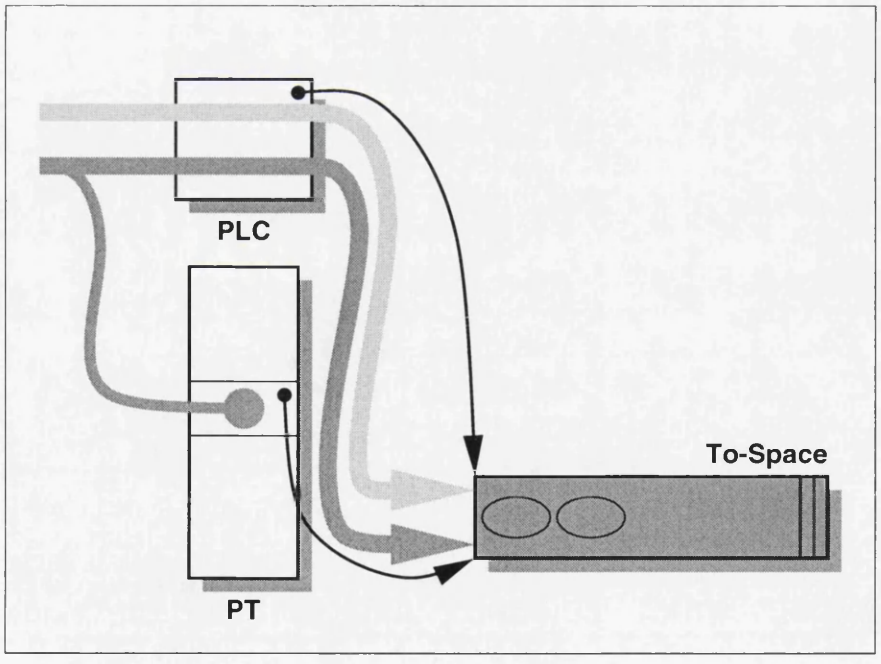


Figure 6.5: Compacting Garbage Collection — Thread Synchronisation — Finish

Figure 6.2 shows the initial state of the system. The partition that is about to be garbage collected is being accessed by reader and writer threads. These have to perform a PLC look-up in order to determine the location of the partition. Additionally, the writer threads have locked the partition for updates by increasing the updating count in the corresponding PT entry (↝§4.8.5). As the partition is locked, the GC thread will not be able to start operating over it, before each writer thread has committed all of its updates.

Figure 6.3 illustrates the moment when the GC thread is given exclusive access to the partition and sets the garbage collection flag (↝§4.8.5). This blocks all writer threads, which first have to ensure that they increase the updating count in the PT entry before accessing the partition[2]. This cannot happen while the garbage collection flag is set. Writer threads are only blocked from accessing this one partition; they are still allowed to access other partitions in the store. Notice that all of the reader threads are still accessing the from-space of the partition without being blocked (this is possible as the GC does not modify from-space).

After the GC finishes its operation, it is necessary for any threads that need to access the partition to be redirected to its new location. This is illustrated in figure 6.4. All reader threads that are trying to access the partition are blocked at the PLC; this ensures that no new reader threads can access it. Then, the GC thread waits until the pinning count of the PLC entry corresponding to the partition is decreased to 1; this ensures that no other threads are accessing it apart from the GC thread itself. Now, it is safe to update the partition location on the PT and PLC entries. During the above operation, threads that are accessing other partitions are not blocked, but are operating normally.

The completion of the garbage collection operation is illustrated in figure 6.5. Once the PT and PLC entries have been updated with the new partition location, the threads waiting at the PLC are unblocked and the garbage collection flag in the PT entry is unset. This allows all threads that need to access the partition to resume their operation and any writer threads to increase the updating count in the PT entry and start updating the partition.

6.3.1.2 Fault-Tolerance Considerations

The adoption of the two-space garbage collection scheme, which migrates objects to the to-space while leaving the from-space unchanged, considerably simplified the introduction of fault-tolerance into the algorithm. The way this was achieved was partly inspired by the *Ghosted Allocation* promotion algorithm (described in section 5.2). To ensure atomicity, a garbage collection takes place in terms of a history (↝§4.4.1). However, all the disk transfers to to-space are applied with synchronous unlogged disk writes that need to be completed before the corresponding history has been committed. The only updates that need to be logged are the following four.

❶ **Allocation of To-Space** — This involves setting the bits of the allocation bitmap to denote that the BBs that comprise to-space have been allocated (↝§3.8.3). A single log record is generated to reflect this update.

❷ **Update of the PT Entry** — The partition location on the PT entry needs to be updated to point to to-space, as illustrated in section 6.3.1.1. A single log record is generated to reflect this update.

❸ **Reference Counts Updates** — The reclamation of garbage objects might cause the reference counts of other objects, directly referenced by them and residing in different partitions, to be decreased. Such

[2] A writer thread will lock the PT entry of a partition only once, since the history mechanism (↝§4.4.1) keeps track of which partitions have been changed for each history.

updates are buffered with the use of the RCUB, described in section 4.10. Each flush of the RCUB generates a single log record.

❹ **De-Allocation of From-Space** — Likewise, this involves unsetting the bits of the allocation bitmap to denote that the BBs comprising from-space have been de-allocated (⇝§3.8.3). A single log record is generated to reflect this update.

The operation of the algorithm, as described above, is guaranteed to be atomic. The marking phase is fault-tolerant by default as it applies no updates to the partition. Therefore a crash during it will only discard any marking the GC has performed and this will have to be restarted upon the next activation of the store. If a crash occurs *during* the sweeping phase, upon startup all the work performed by the GC is discarded, to-space is considered de-allocated, and the partition is reverted to its original location, where it resides unchanged. Additionally, if a crash occurs *after* the corresponding history has been committed then to-space is guaranteed to already reside on disk, because of the synchronous disk writes, and the above updates can be redone, if necessary, by scanning the log. The main strength of the scheme described here is that it ensures fault-tolerance, while requiring a minimal amount of log traffic.

During garbage collection, the BBs that comprise both from-space and to-space must be considered allocated and those of from-space can only be re-used to satisfy a partition-allocation request *after* the GC history has been committed (otherwise, it would not be possible to revert to from-space after a failure).

There is an exception to the above description of the GC, namely when the partition that is being garbage collected contains only garbage objects. When this happens, the partition is de-allocated at the end of garbage collection and there is no need to allocate BBs for to-space. However, the garbage objects in from-space still need to be scanned for the purpose of decreasing reference counts. In this case, the updates that need to be logged are *(i)* the PT entry is updated to be marked free, *(ii)* the partition BBs are de-allocated, and *(iii)* any reference counts that are decreased. Again, it should be pointed out that the BBs comprising the partition can only be used to satisfy a partition request *after* the GC history, and hence the partition de-allocation, has been committed.

6.3.2 Main-Memory Data Structures

This section enumerates the main-memory data structures needed by the compacting GC. Section 6.3.2.1 describes the bitmap, used for object-marking purposes, section 6.3.2.2 describes the stack, used to keep track of the object traversal position, and section 6.3.2.3 describes the object-to-indirectory entry table, used to cache some information about the objects that are visited by the GC, to avoid them being refetched from disk.

6.3.2.1 Marking Bitmap

The GC uses an main-memory bitmap in order to mark objects. If a bit is set then the corresponding object has been marked as live, otherwise it is considered to be garbage. There is one bit per indirectory entry and, since the maximum number of indirectory entries is bounded (currently 32K), the maximum size of this bitmap is also bounded (currently $32K/8 = 4K$). This is a better scheme then having the bits of the bitmap denoting the beginning of objects in the object space, since

- it imposes a much smaller bitmap size (otherwise the bitmap would have to cover the entire object space, which is typically larger than the indirectory) and

- the bit in the bitmap that corresponds to an object can be deduced from the object's PID, as a PID contains the indirectory index of the object (↝§3.7.1); in this way, given a PID, no indirectory entry access is necessary to mark an object or check whether an object is marked.

6.3.2.2 Request Stack

During the marking phase, the indirectory is scanned and every object with a reference count of 1 or greater is considered to be a root and marked as live. The transitive closure of the reachable objects from each root is visited (this scan stops when an object outside the partition has been reached — this can be efficiently deduced from its PID) and these objects are also marked live. One way to perform this traversal would be to use recursion. However, recursion is typically expensive both space-wise and time-wise, therefore a stack is used instead. Each stack entry is simply the PID of the next object which should be visited. Past experience [Pri96, Ham97] has shown that this stack can be kept short, even for large stores or partitions. However, the largest size it can grow to is equal to the maximum number of objects in the partition (currently 32K).

6.3.2.3 Object-to-Indirectory Entry Table

According to the partition organisation of Sphere (↝§3.6.2 and §4.5.2), it is possible to reach efficiently an object's contents from its indirectory entry but not vice versa. However, the sweeping phase of the GC[3] iterates over the objects and, as it relocates each live object to its new position, it needs to access its indirectory entry in order to update the location field in it. The two solutions which are typically adopted in similar situations are not applicable.

- During the marking phase, store in each object a pointer to where its indirectory entry is (can use a word on the object and store its value in the location field of the indirectory entry — a similar scheme has been implemented in the Classic JDK garbage collector). However, this would perform updates to live objects in from-space which are not allowed (↝§6.3.1.2).
- Rather than sweeping over the objects, sweep over the indirectory; this way the problem does not arise. However, this might change the order in which the objects are visited, and can cause the complication discussed in section 6.3.5.

To deal with this problem, an main-memory table is managed, called *Object-to-Indirectory Entry Table* (OIET). Each OIET entry contains three fields:

1. the indirectory entry offset,
2. the object offset, and
3. the next field of the newly-constructed indirectory free-list (see section 6.3.4.5 for more details).

The size of each OIET entry is 12 bytes. Since the maximum number of objects inside a partition is currently $32K$, the maximum size of this table is $32K \times 12 = 384K$. After entries for all objects have been inserted (this is done during a pass after the marking phase, ↝§6.3.4.5), the OIET is sorted according to the object offsets. This provides an efficient map from object offsets to the corresponding indirectory entries.

[3]The sweeping phase iterates over the objects, copies the live ones to the new partition, and decreases the reference counts of any objects reachable from the garbage ones. This definition might be at variance with the one usually used in the literature.

Entries in this table are inserted for all indirectory entries, not only for the ones that correspond to objects. The reason for this is that the sweeping phase also has to visit garbage objects in order to decrease reference counts of the objects they reference, as well as all of the freed indirectory entries in order to insert them in the indirectory free-list. This is described in more detail in section 6.3.4.7.

6.3.3 Analysis

Once the GC has determined the number of live objects (after finishing the marking phase), it has to determine whether it is worth completing the garbage collection or not. If the percentage of live objects is high (this threshold is currently set to 95%), carrying on with the garbage collection will generate too much necessary traffic (most of the objects will need to be copied) with minimum pay-off (not much space will be reclaimed). Therefore, in this case, the GC is trivially aborted, as the marking phase has performed no updates to the partition.

Another possibility is that there are no live objects in the partition. In this case the partition should be de-allocated. However, the garbage objects still need to be scanned for the correct management of reference counts, therefore garbage collection cannot simply abort. Instead, it carries on in a limited mode and does not allocate to-space, since the partition will be de-allocated rather than copied.

The analysis stage is also the one which determines whether the partition needs to be resized. If the object space has reached the indirectory but the indirectory is not full, it might be beneficial to allocate a larger space for the partition. Alternatively, if the indirectory is full but there is still space in the partition, it might be beneficial to allocate a smaller space for the partition.

6.3.4 Detailed Description of the Compacting Garbage Collector

This section gives a detailed description of the Sphere compacting garbage collector. Sections 6.3.4.1 to 6.3.4.7 describe the seven steps in which the GC operates and how it uses main-memory data structures, described in section 6.3.2. Then, section 6.3.4.8 summarises this section.

6.3.4.1 Step A — Marking Phase Initialisation

This is the first step of the garbage collection. It initialises the marking bitmap and the request stack and prepares the marking phase. It does not need to access the partition header, as it can find the partition size (and, hence, the beginning of the indirectory) from the PLC (↝§4.9.2).

Some of the GC steps will be illustrated with a concrete example. The initial phase of this example can be seen in figure 6.6. The partition contains four objects **A**, **B**, **C**, and **D**, at offsets *a*, *b*, *c*, and *d* respectively, and five indirectory entries, one of them being free and having been inserted in the indirectory free-list. Only object **C** has a reference count greater than 0. After Step A, the marking bitmap, which has one bit per indirectory entry, has been initialised with all its entries having been set to 0.

6.3.4.2 Step B — Marking Phase (Pass 1)

During the first pass over the data, the objects that are reachable are marked, using as roots all objects with reference count greater than 0. The marking phase works as follows.

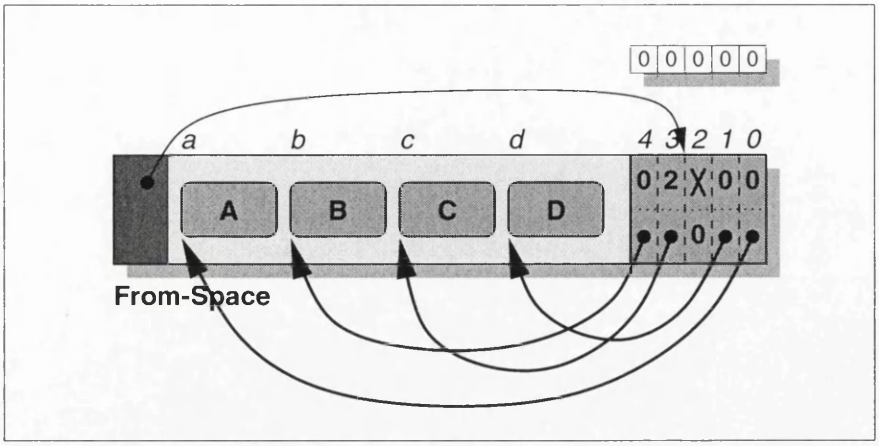


Figure 6.6: Compacting Garbage Collection — After Step A

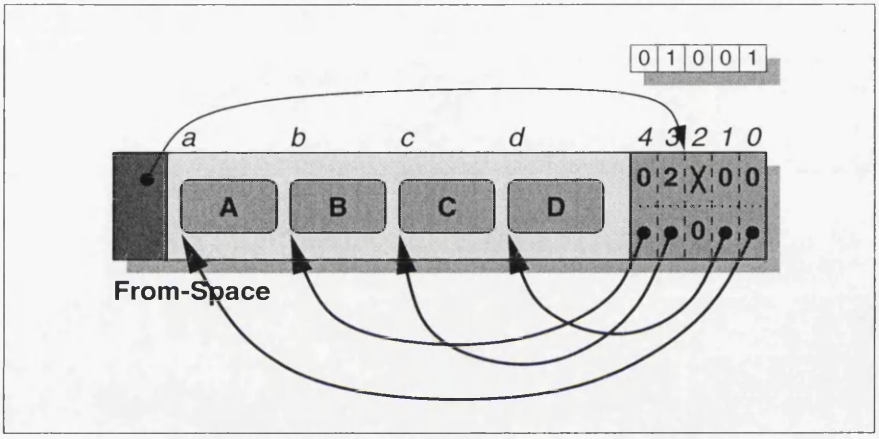


Figure 6.7: Compacting Garbage Collection — After Step B (Pass 1)

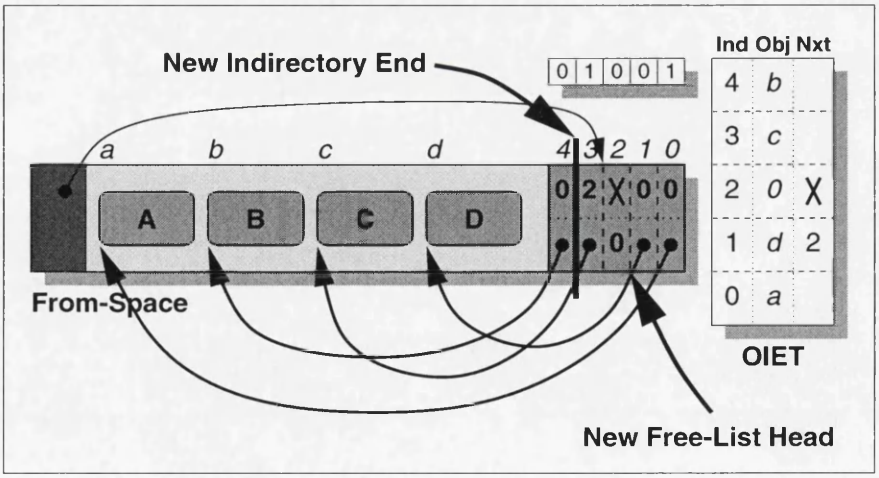


Figure 6.8: Compacting Garbage Collection — After Step E (Pass 2)

- The indirectory is scanned and the PID of each entry which *(i)* has reference count greater than 0, *(ii)* has offset field different from 0 (i.e. it has not been freed)[4], and *(iii)* is not already marked, is marked and pushed onto the request stack.
- For every PID popped from the stack, the corresponding object is fetched and its pointers are identified. Every object reachable from it which *(i)* is not already marked and *(ii)* is contained in the same partition is pushed onto the stack.
- For every object which is pushed onto the stack, a counter is incremented which keeps track of the number of live objects inside the partition.
- The scanning phase is terminated when *(i)* all indirectory entries have been visited and *(ii)* the request stack is empty.

Figure 6.7 illustrates the state of the partition of the example after Step B. The root of the partition is **C**, as it is the only object with a reference count greater than 0. After the marking phase is complete, it turns out that only objects **A** and **C** are live, with the other two being garbage; this is reflected on the bitmap. The use of the request stack is not illustrated in the figure.

6.3.4.3 Step C — Analysis

This step decides whether it is worth continuing garbage collecting the partition. It compares the number of live objects with the total number of objects inside the partition. There are three possible outcomes.

- **Live Objects > Threshold** — In this case the GC aborts and no further work is carried out.
- **No Live Objects Found** — In this case, the partition should be de-allocated so the space it is taking up can be re-used. However, the GC still needs to scan the garbage objects in order to decrease the reference counts of any reachable objects in other partitions. Therefore, the GC continues in a limited mode.
- **0 < Live Objects ≤ Threshold** — This is the "normal" operating mode of the GC and will move the partition to a different location in the store.

6.3.4.4 Step D — Sweeping Phase Initialisation

This step initialises the sweeping phase. It calculates the new indirectory end by scanning the bitmap backwards and stopping at the first live entry. It also initialises the OIET and sets the new indirectory free-list head to null.

6.3.4.5 Step E — Indirectory Preparation (Pass 2)

This step scans the indirectory, populates the OIET, and calculates the next fields of the new indirectory free-list (which are temporarily stored in the same table). The scan takes place backwards in the indirectory (i.e. forward in the partition) so that the free-list grows backwards into the indirectory. This means that entries which are towards the beginning of the indirectory will be allocated before the ones which are at the end. This has the potential to eventually create a block of free entries at the end of the indirectory and

[4]When freed indirectory entries are added to the indirectory free-list, their reference count field is used for linking purposes and their offset field is set to 0 (↝§4.5.3). Therefore, an indirectory entry with a reference count greater than 0 does not necessary correspond to a live object and the offset field also has to be checked.

remove them by just shrinking it.

For every indirectory entry visited:

- The indirectory index and object offset are inserted in the OIET. This also occurs even if the entry is already freed and does not correspond to an object (in this case the object offset is 0).
- If the entry is not set in the bitmap (i.e. it is either an already freed entry or corresponds to a garbage object) the head of the new free-list is saved in the next field of the OIET entry and the new head points to the current entry. However, this does not happen if the entry is located beyond the new indirectory end. In this case the entry does not need to be inserted in the new free-list since it will be reclaimed.

Figure 6.8 illustrates the state of the partition of the example after Step E. The new indirectory end (calculated during Step D) points to entry 3, which is the last marked entry; this implies that entry 4 will be reclaimed, since the indirectory will shrink. The OIET contains five entries, one per indirectory entry, mapping them to the location of the corresponding object, apart from entry 2, which was originally freed and its offset field is 0. The entries appear in reverse order, since the scan over the indirectory took place backwards. Finally, the head of the new indirectory free-list points to entry 1, which in turn points to entry 2 (as seen in the next field of the OIET).

6.3.4.6 Step F — To-Space Preparation

During this step, the space for the new location of the partition is allocated, if necessary (i.e. if there is at least one live object in the partition). The header of the partition is initialised and the indirectory pages, up to the new indirectory end, are copied from the old space to the new one. Even though the object locations and free-list next fields will be set during the next stage, copying the indirectory is still necessary since it copies the reference counts and LSNs. Finally, the OIET is sorted according to the object locations. This allows the next step to sweep over the objects sequentially and copy them in the same order as the one they appear in in the partition.

Figure 6.8 illustrates the state of the partition of the example after Step F. The new space has been allocated for it, the new header initialised, and the new free-list head (which points to entry 1) set up. The indirectory has also been copied. Finally, the OIET is sorted according to object location.

6.3.4.7 Step G — Sweeping Phase (Pass 3)

This is the last and most complicated phase of the garbage collection. It copies all the live objects from from-space to to-space and sets up their indirectory entries. During this step the OIET, which is sorted according to object location, is scanned and for each entry the corresponding bit in the bitmap is examined.

- If the bit is set, the object is copied to to-space and the object location in its indirectory entry is updated accordingly.
- If the bit is not set, it either means that the object is garbage (object offset field is not 0) or the entry does not correspond to an object (object offset field is 0). In the former case, the object's references must be scanned in order to decrease appropriately the reference counts of objects in other partitions reachable from it. In either case, the object location in the indirectory entry is set to 0 and the free-list next field is copied from the table entry to the reference count field on the indirectory entry. This last step only takes place if the indirectory entry is located before the new indirectory end.

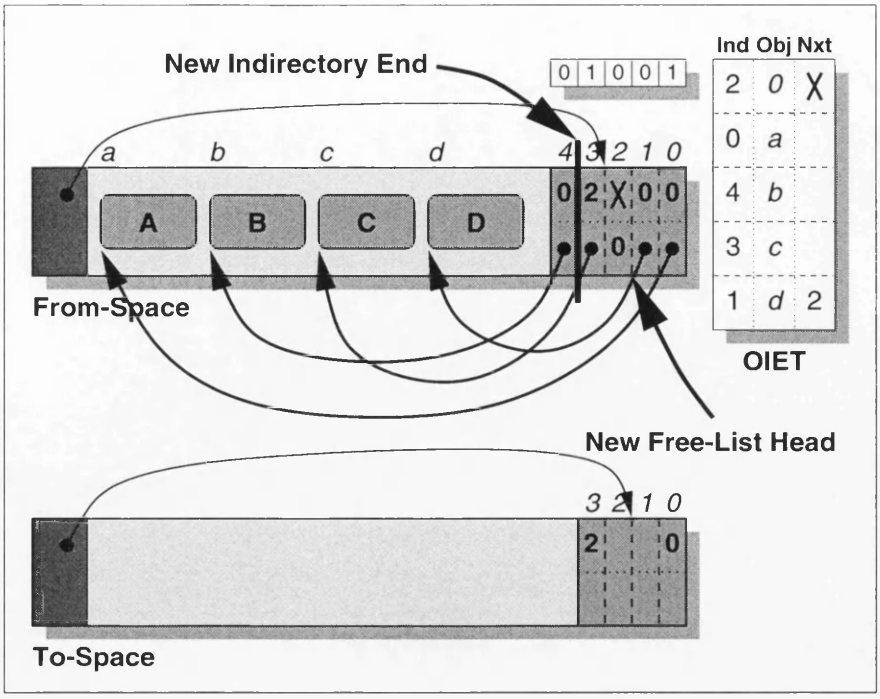


Figure 6.9: Compacting Garbage Collection — After Step F

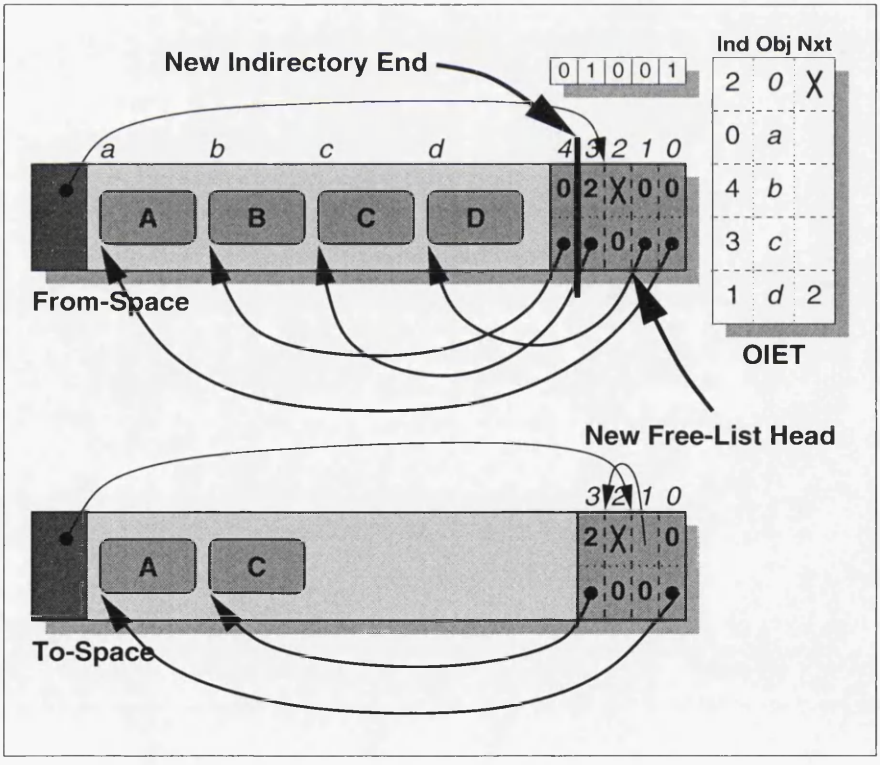


Figure 6.10: Compacting Garbage Collection — After Step G (Pass 3)

The last two steps of this phase are *(i)* to update the partition location on the partition table to point to the new space and *(ii)* to notify any threads waiting to access the partition than they can now do so, as described in section 6.3.1.1. The garbage collection is then committed by applying and logging the updates enumerated in section 6.3.1.2.

Figure 6.10 illustrates the state of the partition of the concrete example after Step G. At this point, the partition is ready for use with both live objects having been copied to to-space, their indirectory entries set up, and the indirectory free-list reconstructed.

6.3.4.8 Summary

Table 6.1 summarises the operation of the Sphere compacting GC and illustrates which data structures are touched by each GC step. It is important to notice that no writes to from-space take place. As mentioned earlier in this chapter, this contributes towards the fault-tolerance of the algorithm and also the ability of any reader threads to read from-space concurrently with the operation of the GC without further synchronisation (apart from the operation described in section 6.3.1.1). Also, even though the GC operates in seven steps, only two of them (B and G) touch the object space, the rest concentrating on the two smaller spaces (indirectory or partition header).

Note that there are no GC steps that use both the request stack and the OIET, therefore the space allocated for the stack can be re-used for the OIET.

6.3.5 Object Order is Important

The two small-object partition regimes that have been implemented in Sphere (↝§4.13.2) retain the invariant that an object (this refers to the object contents and header, not indirectory entry, ↝§4.5.4) entirely resides on a single page. This ensures that a single I/O operation will be sufficient to fetch its contents. Given this, most pages have some unused space at the end, where an object did not fit and was allocated in the following page.

To facilitate the object sweeping operation, this unused space has been "disguised" as an object, by setting the kind field in its header to a reserved value and its size to reflect the size of the unused space. Such objects are referred to as *Empty Objects*. However, the introduction of empty objects has introduced the paradox that, if during garbage collection the objects in top-space are not allocated in the same order as they are in from-space, to-space might run out of space, even if it is of the same size as from-space.

Imagine that there are only two kinds of objects allocated in the same partition: **A** and **B** of size α and β respectively, with $\alpha \gg \beta$ and $\alpha + \beta = \textit{page size}$. If the allocation pattern is **A**, **B**, **A**, **B**, **A**, **B**, ... then each page of the partition will contain one object of type **A** followed by one object of type **B**. However, if the allocation pattern changes to **A**, **A**, **A**, ..., **B**, **B**, **B**, ... then each page will contain one object of type **A**, with the rest of the space being filled by an empty object, but there will not be any space for the objects of type **B**. If the allocation order remained the same, this problem would not occur.

6.3.6 Discussion

This section described the Sphere compacting garbage collector, how it operates, and how it is synchronised with mutator threads. Its advantages include allowing mutator reader threads to operate concurrently over

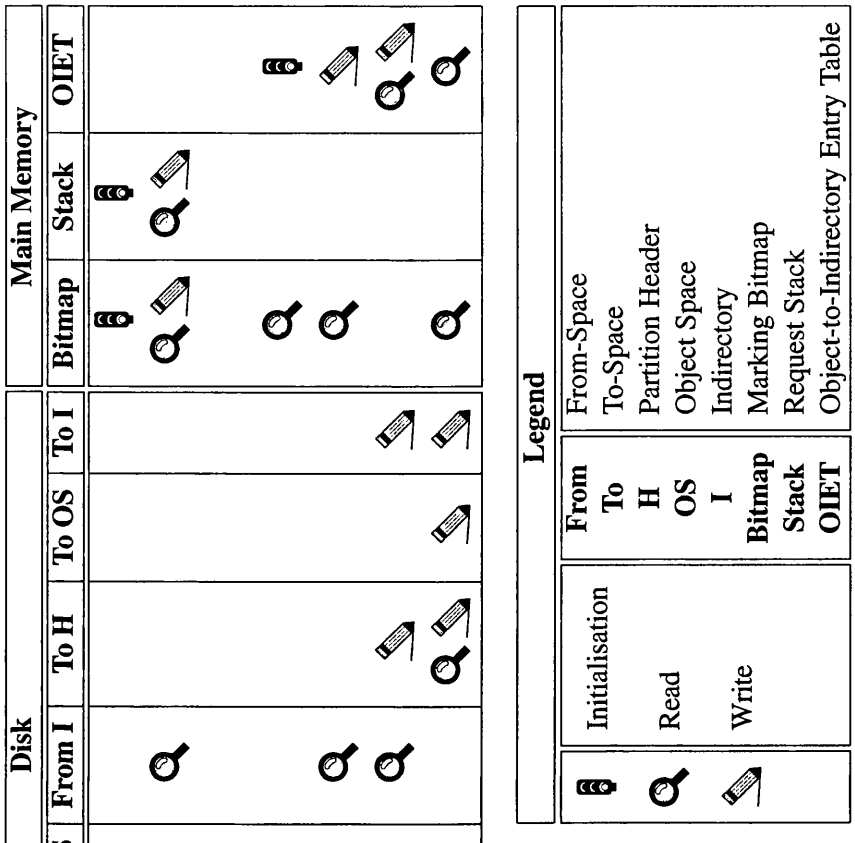


Table 6.1: Summary of the Operation of the Sphere Compacting Garbage Collector

the same partition, attempting to minimise the disk accesses it performs, and compacting the relevant partition, in order to avoid fragmentation problems.

However, the algorithm also has two main disadvantages.

- Since a mutual exclusion between the GC and writer threads is assumed, writer threads may have to block waiting for the GC operation to complete.
- Since the GC relies on cross-partition reference counts to identify roots into the partition, cross-partition garbage cycles will not be reclaimed.

Future work to deal with the above in discussed in section 6.8.

6.4 Measurements

This section presents measurements taken from an initial prototype implementation of the Sphere compacting garbage collector, which was described in section 6.3. The benchmarks used were modified versions of MultiBench, which was described in section 4.3. Section 6.4.1 deals only with reader threads and explores the effect that the sleep time between garbage collections and the percentage of garbage objects in partitions have on the overall performance of the system. Section 6.4.2 introduces a writer thread that creates garbage dynamically and focuses on GC elapsed times, their breakdown, and the synchronisation between the GC thread and the writer thread. Section 6.4.3 briefly discusses the results.

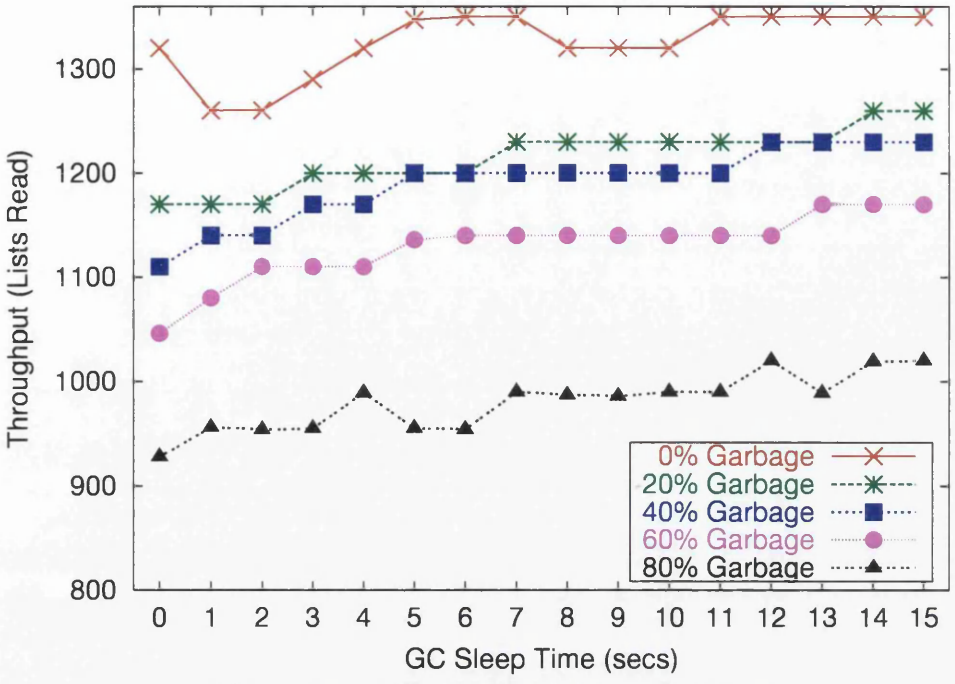
In all the measurements, no special heuristics were employed when choosing a partition for garbage collection and the GC thread simply iterated over the allocated partitions in the store. Also, unless otherwise specified, the configuration of the system was the same as the one described in section 4.3.

6.4.1 Readers-Only Benchmark

In this section, a measurement run involved 30 threads concurrently iterating over lists chosen at random. Each run lasted 5 minutes and the throughput, i.e. the overall number of lists read by all reader threads, was calculated. The two parameters that were varied were the following.

- **GC Sleep Time** — this is the time the GC thread slept between garbage collections. This was varied from 0 to 15 secs.
- **Garbage Percentage** — the percentage of garbage objects in each partition. The values chosen for this were 0%, 20%, 40%, 60%, and 80%. It must be emphasised that the number of live objects in the store did not vary (i.e. there always were 500 lists with 1,000 nodes each). However, it was arranged that all list nodes were allocated to partitions with the required amount of garbage objects already allocated in them. It follows that increasing the percentage of garbage objects in the partitions also increased the number of partitions needed to accommodate the lists, which was 20, 24, 32, 48, and 95 respectively (all partitions sizes were of size 1MB).

Figure 6.11 illustrates how the percentage of garbage in the partitions and the GC sleep time affect the throughput of the benchmark. The throughput values *without* the GC thread running are also given as reference points. The figure shows that the introduction of garbage objects in partitions decreases the throughput of the benchmark in all cases, even the one without the GC thread running. This is caused by the increased



Throughput without the GC thread running:

0%: 1350 **20%:** 1320 **40%:** 1290 **60%:** 1230 **80%:** 1080

Figure 6.11: Readers-Only Benchmark — Throughput

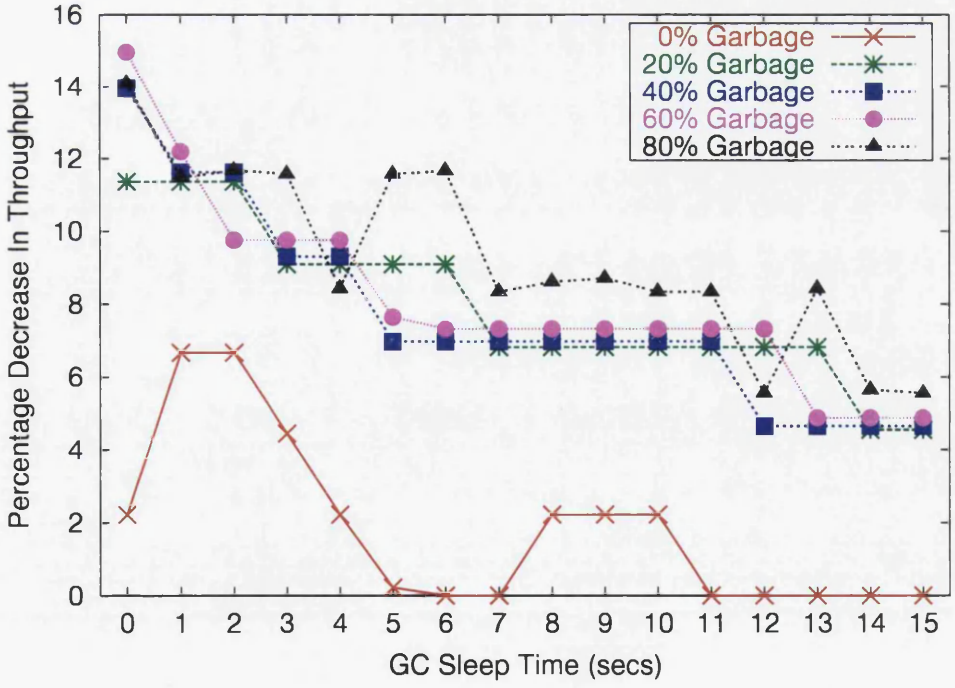


Figure 6.12: Readers-Only Benchmark — Percentage Decrease in Throughput

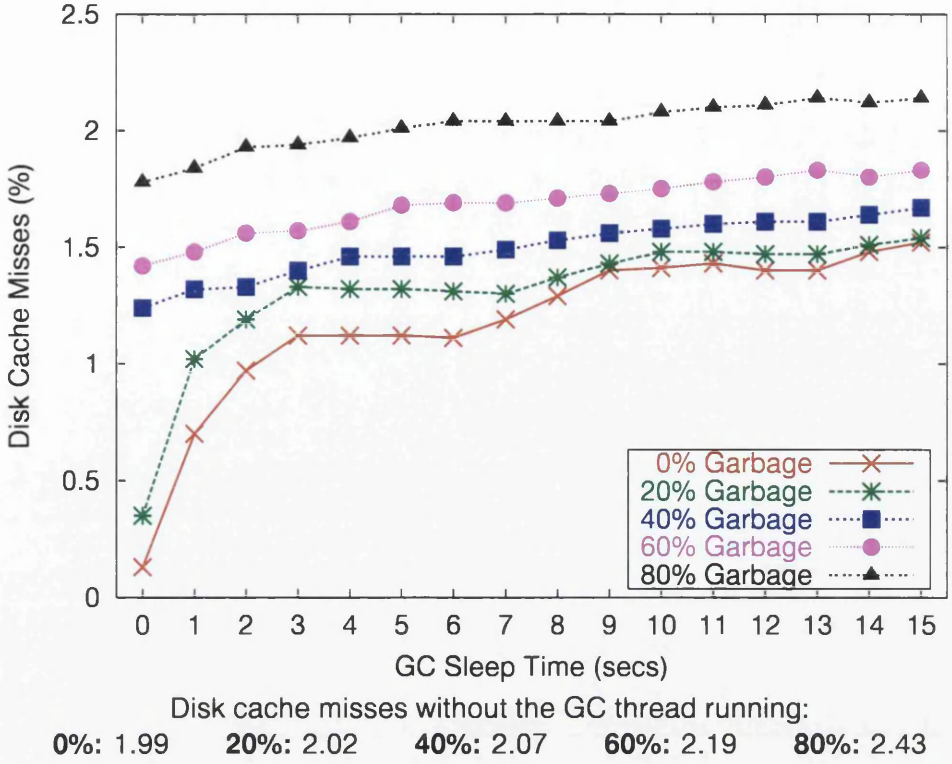


Figure 6.13: Readers-Only Benchmark — Disk Cache Misses

number of disk-cache misses, as will be described below. The increase of the GC sleep time increases the throughput in most cases, apart from the runs with 0% and 80% garbage, whose behaviour is slightly more erratic.

Figure 6.12 presents the data in figure 6.11 as percentage decreases over the reference values (with no GC thread running). In most cases, the decrease varies from around 15% to 5% as the GC thread sleep time increases (the rather erratic behaviour of the runs with 80% garbage is also obvious here). The main exception is the runs with 0% garbage which in several cases actually reach the reference throughput and impose no performance decrease. This is due to the fact that, as there are no garbage objects in the store, the GC thread only performs marking phases and does not actually move partitions, hence it performs less work than in the runs with garbage objects.

Finally, figure 6.13 displays the percentage of disk-cache misses for the benchmark runs (the values presented are for all disk-cache accesses during the benchmark, including the reader threads and the GC thread). From the figure it is obvious that the disk-cache misses are affected negatively by the amount of garbage objects in the store. However, surprisingly, the disk-cache misses *decrease* when the GC is invoked more often. This is observed consistently in all configurations. The explanation for this is that in this particular case the GC causes less disk-cache misses than the operation of the reader threads, hence invoking it more often seems to improve the overall disk-cache miss ratio.

6.4.2 Readers-Writer Benchmark

This section contains measurements of a single run of the MultiBench with 30 reader threads and the GC thread running against a store that initially had no garbage objects (i.e. only the list nodes and the object containing the list heads were allocated, spanning over 20 1MB partitions). The GC sleep time was set to one second. However, a writer thread was also launched, which deleted one list head every second, creating garbage objects dynamically. The benchmark was run for 20 minutes, during which the writer thread succeeded in deleting all 500 list heads. During this time, the GC was invoked 309 times, of which *(i)* 261 (84.5%) actually moved the partition, *(ii)* 19 (6.1%) deleted the partition, as it was empty, and *(iii)* 29 (9.4%) only performed the marking phase and deduced that the partition contained too many live objects to continue.

Figure 6.14 shows the distribution of total GC times. The majority (around 75%) lasted less than 3 secs and the average was kept low at 2.4 secs. These results are encouraging. However, the remaining 25% of the times are distributed from 3 up to a maximum of 13.6 secs, which is excessive for 1 MB partitions.

The majority of the GC time was spent in one of the three passes (⇝§6.3.4.2, §6.3.4.5, and §6.3.4.7), the times of which were also analysed. Figure 6.15 includes the distribution of the marking phase times (Pass 1, ⇝§6.3.4.2). 81% of the these, as well as the average, were under 1 sec. The remaining 19% varied between 1 and a maximum of 5 secs. Figure 6.16 shows the distribution of the indirectory preparation phase times (Pass 2, ⇝§6.3.4.5). Notice that there were 260 of them, since not all of the 309 GC invocations actually reached this phase (this is also the case for the sweeping phase). The vast majority of these times (over 95%) as well as the average fell under the 1 sec mark and most of the remaining ones under the 2 sec mark. However, there were three outliers, taking up to a maximum of 6.1 secs (the reasons behind this have yet to be investigated). Finally, figure 6.17 shows the distribution of the sweeping phase times (Pass 3, ⇝§6.3.4.7). 73% of them lasted less than 1 sec and 14%, including the average, lasted between 1 and 2 secs. Most of the remaining 13% are distributed almost evenly up to the 7 sec mark, while one reached a maximum of 7.4 secs. A comparison of this figure with the two previous ones yields that sweeping is the most expensive phase of the garbage collection process, being roughly 2 and 3 times slower that the marking and preparation phases respectively.

There are several factors that affect garbage collection time. The following two were investigated: disk-cache misses and the number of objects in the partition (not only live ones, as the garbage ones also need to be scanned to decrease reference counts, ⇝§6.3.4.7). Figure 6.18 plots the collection times against the percentage of disk-cache misses[5]. From the figure, it is obvious that there is a very strong correlation between them. In fact there seem to be two distinct lines, one for partitions that were moved and one for partitions that were only marked. Additionally, figure 6.19 shows the correlation between the garbage collection times and the total number of objects in the partition. The correlation in this case however is not as strong as in the previous figure.

Figure 6.20 plots the start time of the GC invocation (defined as seconds into the benchmark) against the total number of objects in the partition. The few obvious outliers are the ones corresponding to the last partition in the store, which was not fully-populated. The prominent steps in this figure are caused by reference counts retaining objects, even when the corresponding list head has been made inaccessible by the writer thread. When, consequently, the GC thread accesses the first partition, where the root object containing the list heads is stored, does it have a chance to find some garbage objects and then decrease any reference

[5]The disk-cache misses illustrated in the figure reflect the disk-cache miss percentage during the corresponding invocation of the GC. They refer to all disk-cache accesses and not only the ones by the GC thread.

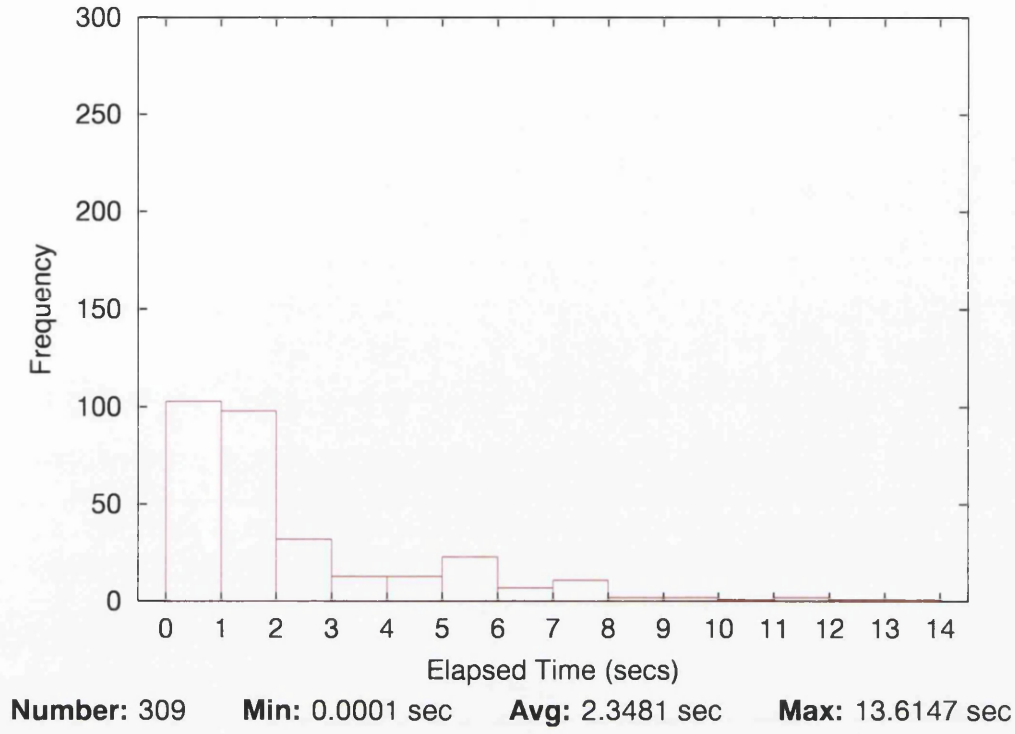


Figure 6.14: Readers-Writer Benchmark — Total GC Elapsed Times

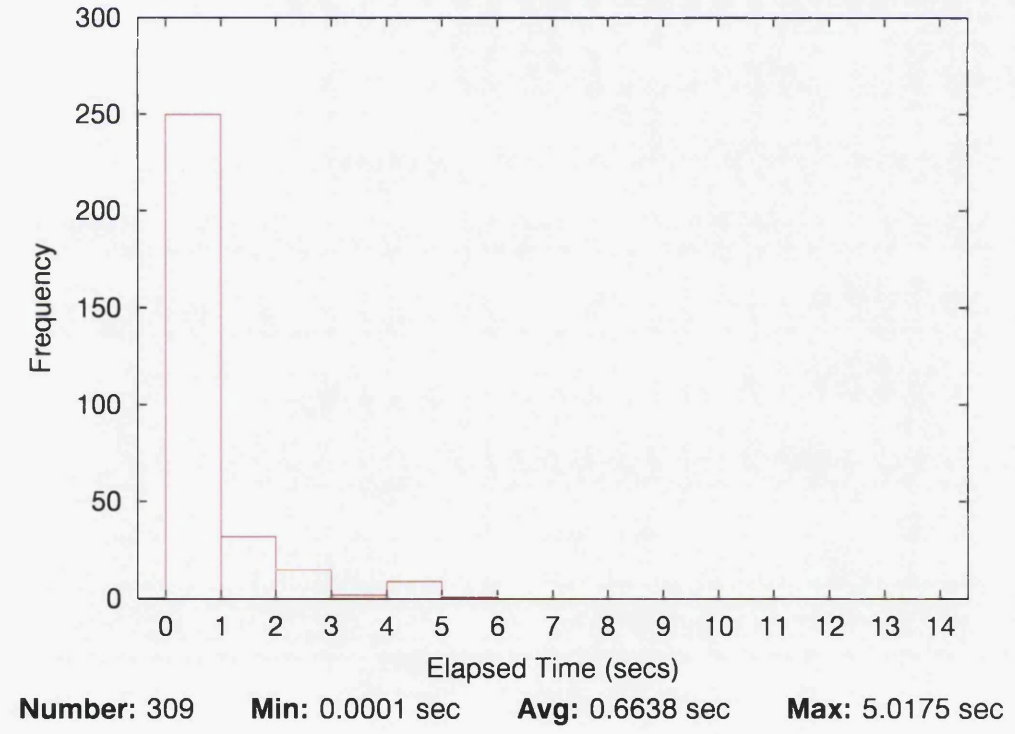
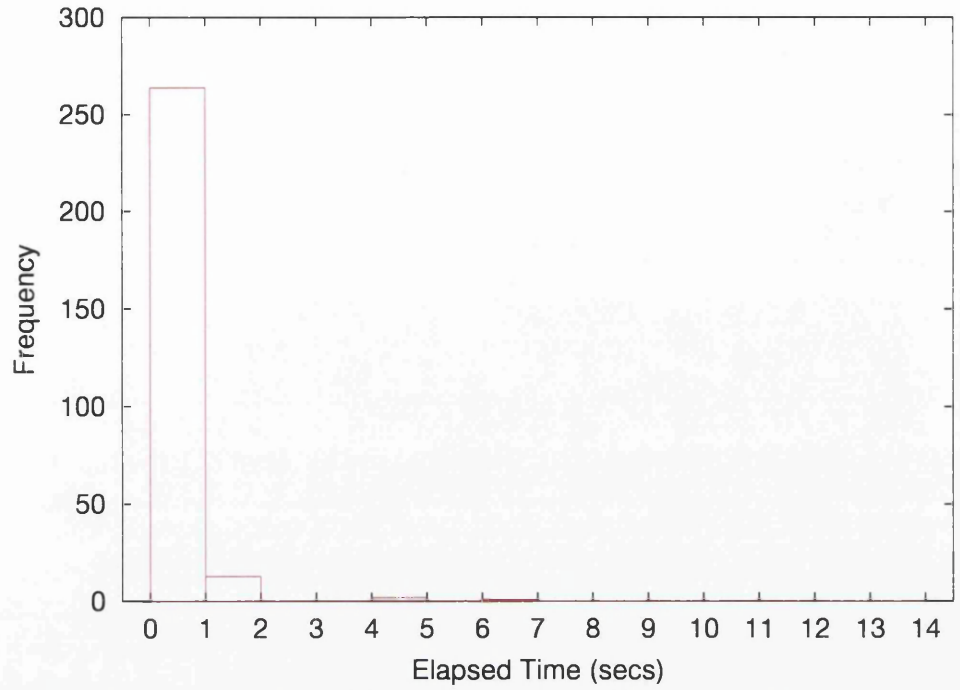
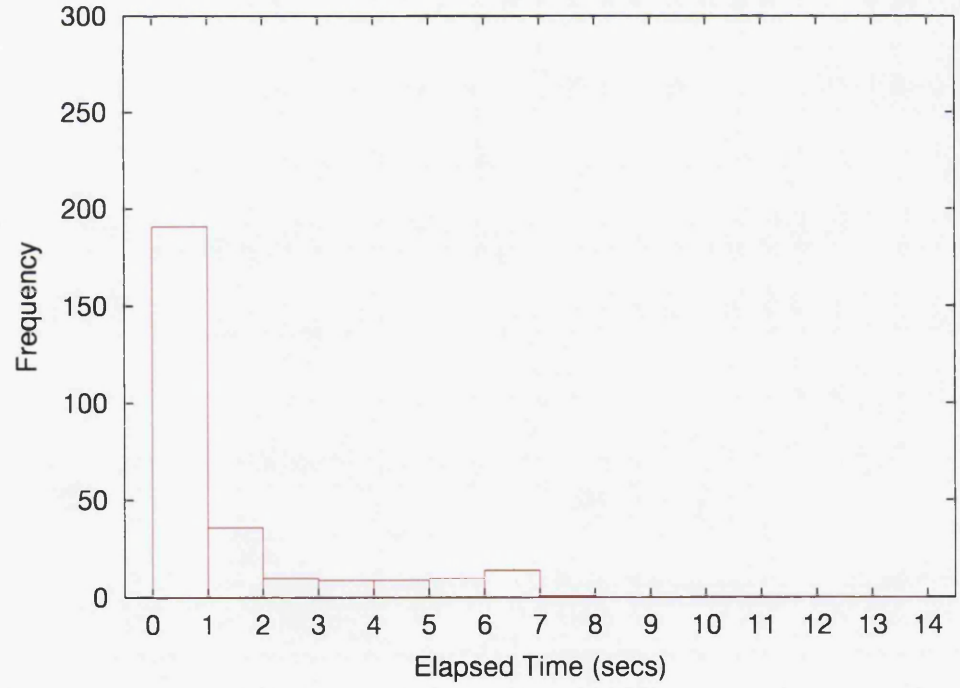


Figure 6.15: Readers-Writer Benchmark — Marking Phase (Pass 1) Elapsed Times



Number: 280 **Min:** 0.0330 sec **Avg:** 0.4377 sec **Max:** 6.0777 sec

Figure 6.16: Readers-Writer Benchmark — Preparation Phase (Pass 2) Elapsed Times



Number: 280 **Min:** 0.0019 sec **Avg:** 1.3466 sec **Max:** 7.4344 sec

Figure 6.17: Readers-Writer Benchmark — Sweeping Phase (Pass 3) Elapsed Times

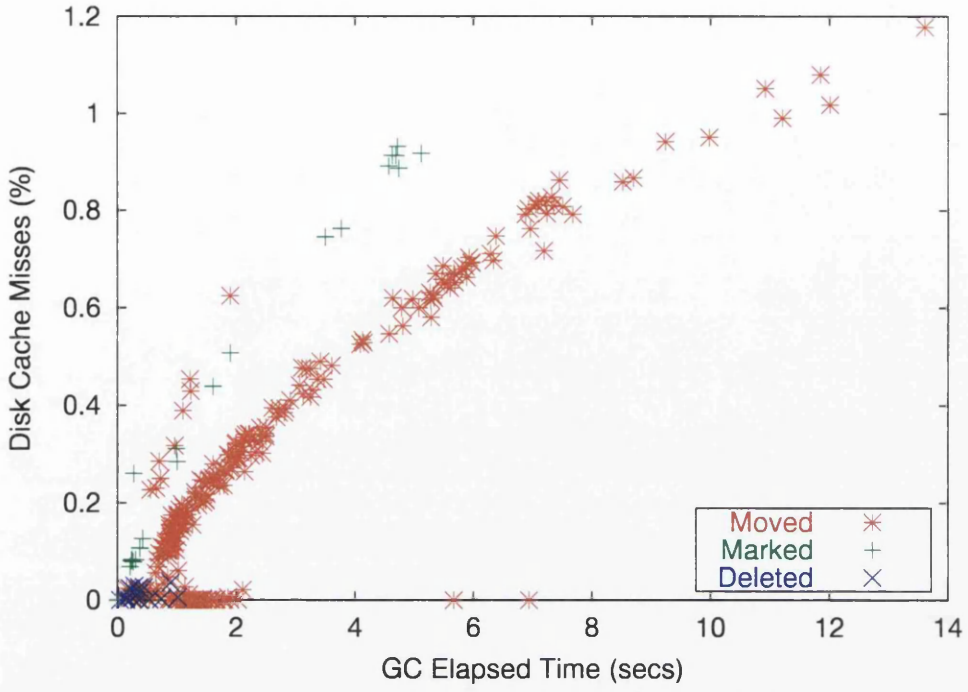


Figure 6.18: Readers-Writer Benchmark — GC Elapsed Times and Percentage of Disk-Cache Misses

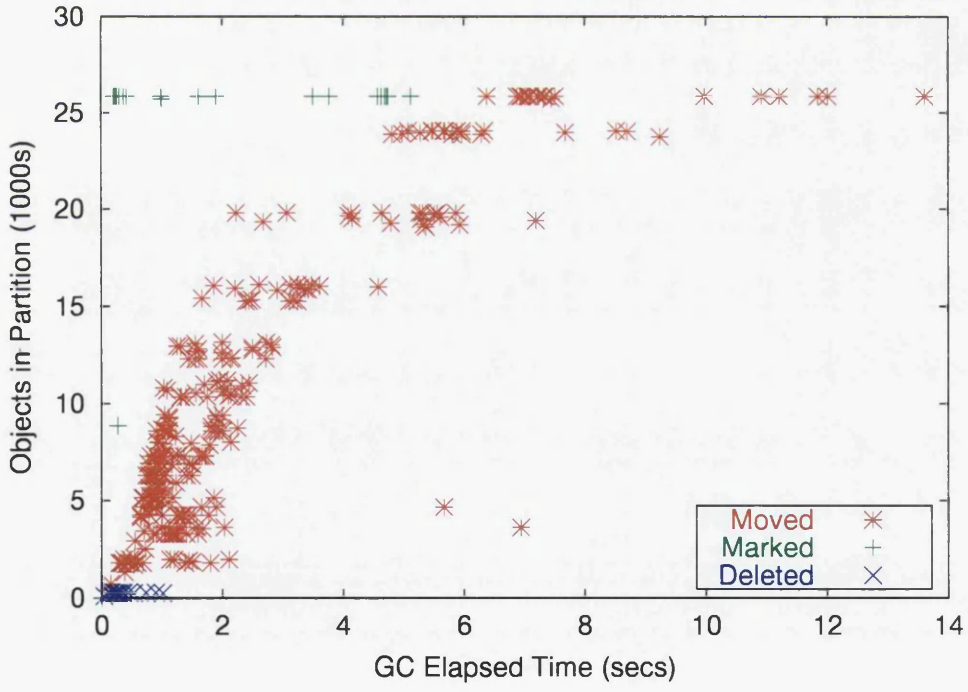


Figure 6.19: Readers-Writer Benchmark — GC Elapsed Times and Total Number of Objects in Partitions

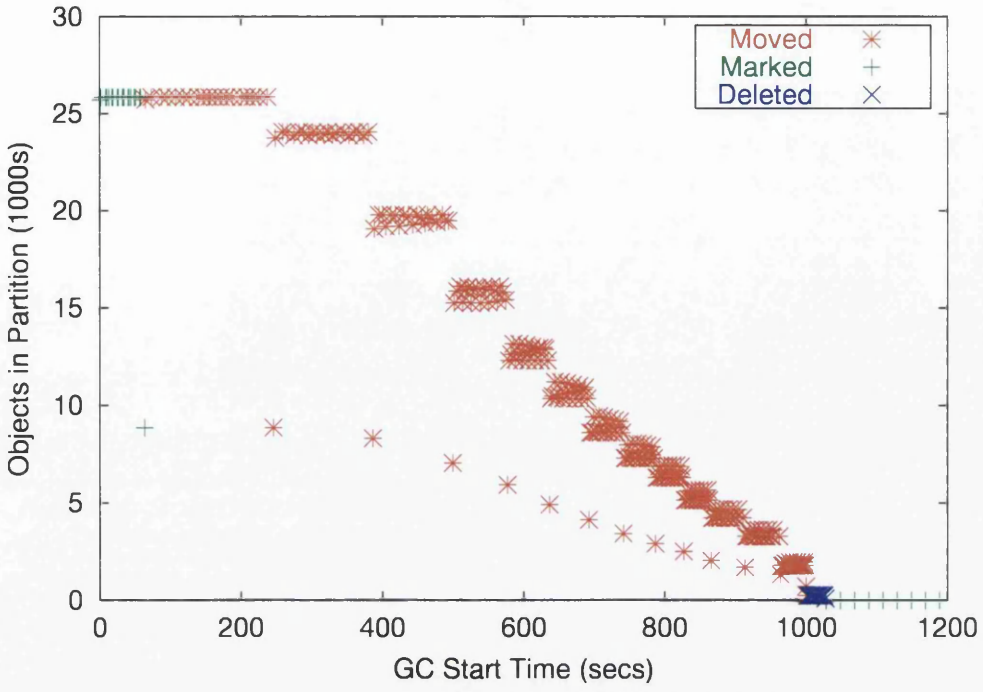


Figure 6.20: Readers-Writer Benchmark — GC Start Times and Total Number of Objects in Partitions

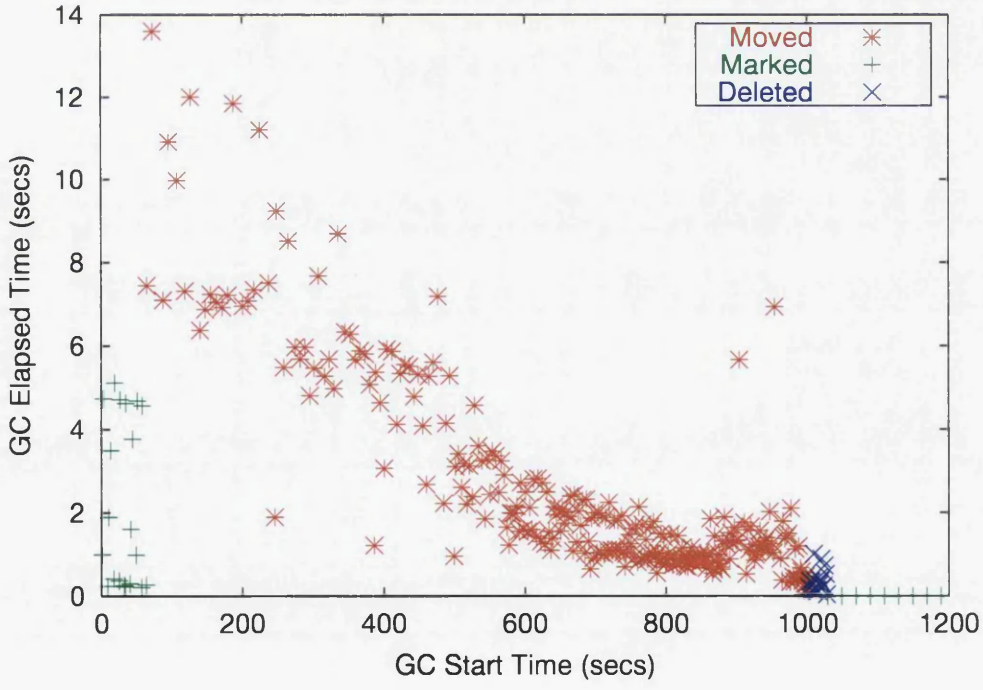


Figure 6.21: Readers-Writer Benchmark — GC Start Times and GC Elapsed Times

counts for references that it holds to other partitions. The size of the steps gets smaller because the GC times decrease when the number of objects in a partition decreases (as already illustrated in figure 6.19). Figure 6.21 plots the start time of the GC invocation against its elapsed time. Given the previous figure it is not surprising that the elapsed times steadily decrease (apart from two outliers) towards the end of the benchmark.

The final set of measurements concentrated on the disruptiveness of the GC on the reader and writer threads. Figure 6.22 shows the distribution of the update thread blocking times, imposed by the mutual exclusion between updates and garbage collection (↝§6.3.1.1). During the benchmark, the update thread performed 500 updates to the store (one per list) and only 10 of them blocked. From these 10, 5 were blocked for under 1 sec and the rest were distributed almost evenly up to a maximum of 7.2 secs (this was probably caused by some of the excessive GC times, presented in figure 6.14).

Figure 6.23 shows the distribution of the GC thread blocking times. Out of the 309 invocations, only 5 of them blocked and all of them were well under one second. However, this was encouraged by the fact that the updates applied by the writer thread were very brief.

Finally, while all reader threads were blocked for the PT and PLC entries to be updated with the new partition location (↝§6.3.1.1), the GC thread waited to achieve exclusive access to the PLC entry for a maximum of 0.03 secs and the reader threads were blocked for a maximum of 0.3 secs (the average was 0.04 secs).

6.4.3 Discussion

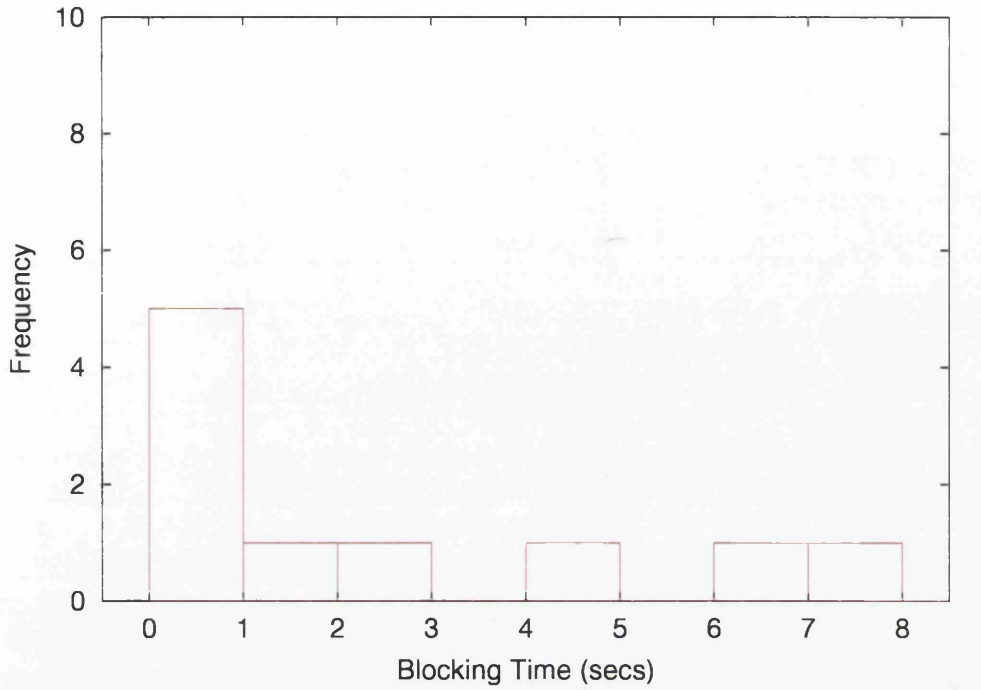
The results obtained here from the prototype implementation of the Sphere concurrent compacting GC are encouraging. All average times obtained were good, the update thread was only blocked 10 times (out of the 500 updates it performed), and the synchronisation times with the reader threads was better than expected. The presence of the GC thread imposed acceptable throughput decreases, from around 15% to 5% depending on the GC thread sleep time, and even no decrease when no garbage objects were present in the store.

However, the most severe problem encountered was the excessive times for some GC invocations (up to 13.6 secs to collect an 1MB partition). This also directly affected the time the update threads stayed blocked during the operation of the GC, which was 7.2 secs in the worst case. Additionally, the low GC thread blocking times due to updates on the partition are also encouraged by the fact that the writer thread was only updating one object in one partition. This is an atypical case.

Future work will be concentrated on determining why the performance of some GC invocations was so degraded and improve on it. Finally, even though microbenchmarks like the ones used here are good in evaluating particular parts of a system, larger applications should also be used in order to evaluate the GC in a more realistic environment (this is discussed further in section 6.9.2).

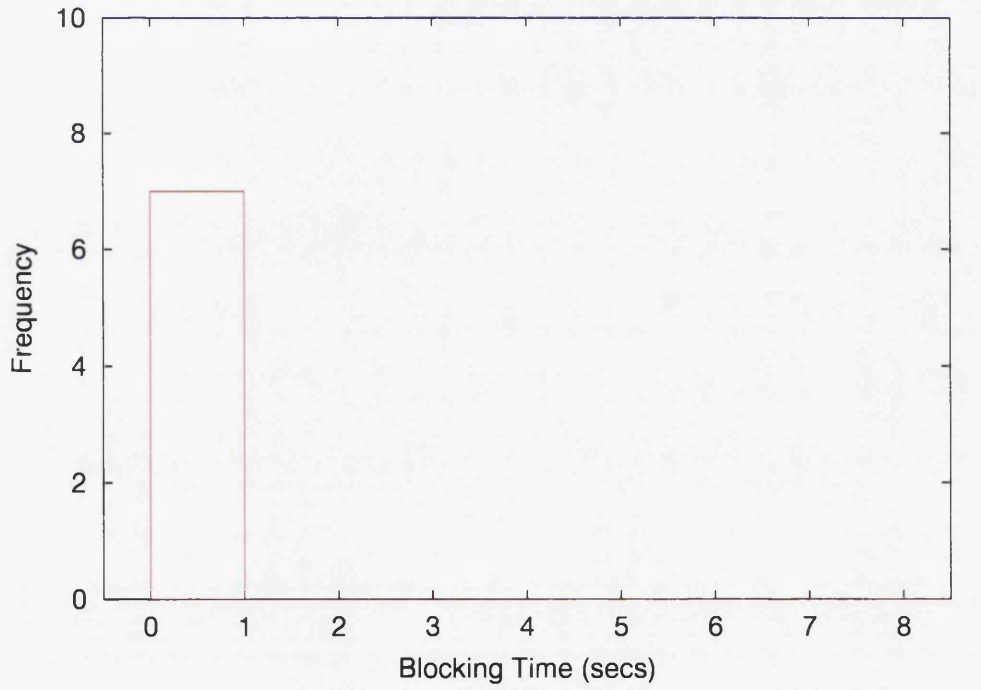
6.5 Interaction with the Mutator

One of the initial requirements for Sphere was to keep the interaction between the store and the mutator to a minimum (↝§3.2). This has mostly been achieved, given that all interaction takes place through the Sphere public API (↝§A) and the implementation of object kinds (↝§3.5.2 and §4.13.1) allows the store to identify the references and the structure of an object, without further interaction with the mutator. However,



Number: 10 **Min:** 0.0600 sec **Avg:** 2.5424 sec **Max:** 7.2304 sec

Figure 6.22: Readers-Writer Benchmark — Writer Thread Blocking Times



Number: 7 **Min:** 0.1100 sec **Avg:** 0.2913 sec **Max:** 0.5600 sec

Figure 6.23: Readers-Writer Benchmark — GC Thread Blocking Times

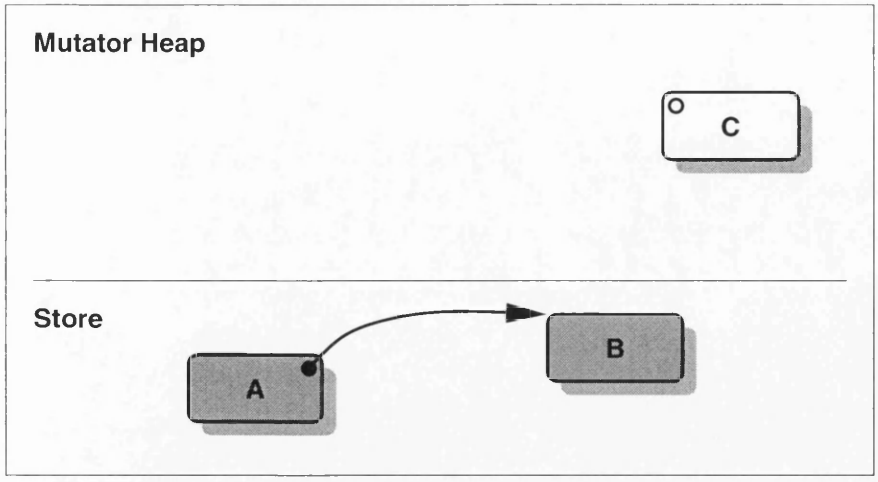


Figure 6.24: Object-Caching Problem — Initial

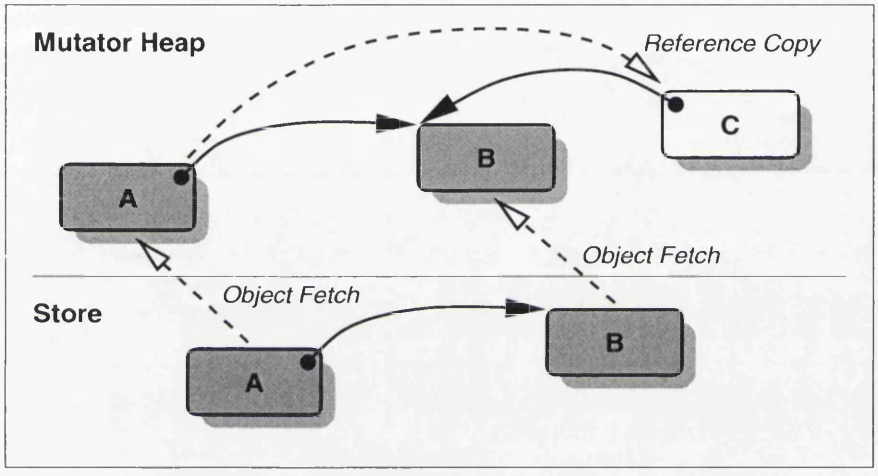


Figure 6.25: Object-Caching Problem — After Fetching the Objects

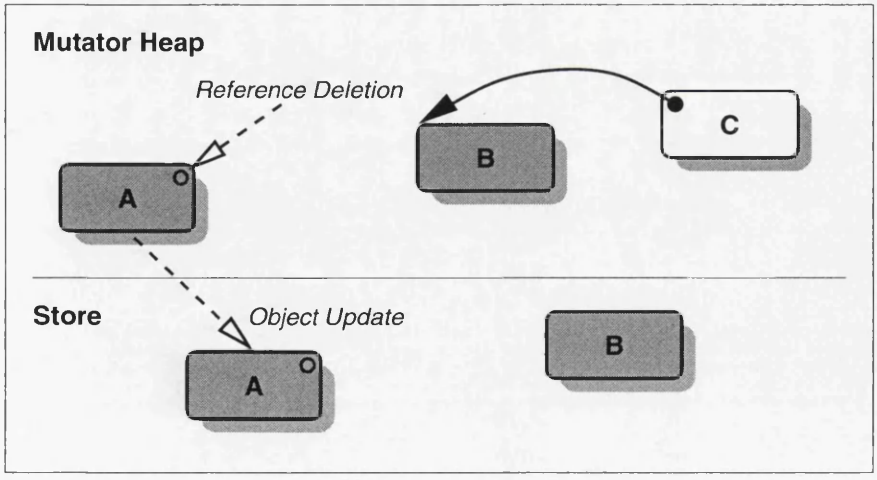


Figure 6.26: Object-Caching Problem — After Deleting the Reference

the presence of a concurrent disk garbage collector introduces the following interesting problem.

Consider the scenario in figures 6.24 to 6.26. In figure 6.24, two objects, **A** and **B**, reside in a Sphere store. Assume that **A** holds the only reference to **B**. Additionally, a third object **C** resides in the mutator heap, which is transient and has no store image (↝§5.1). Figure 6.25 shows the objects **A** and **B** being fetched into the mutator heap and the reference to **B** in **A** (now swizzled — this is not illustrated) copied in object **C**. Finally, figure 6.26 shows the reference field in **A** being deleted and the change propagated to the store. At this point, the only reference to **B** is cached in the mutator heap. If the garbage collector runs without consulting the mutator, it will find no references to **B**, hence might wrongfully reclaim **B**. This is a severe flaw, as the mutator might attempt to propagate updates to object **B**, but instead update the contents of a different newly-allocated object that is using **B**'s indirectory entry or access a free indirectory entry with no corresponding object.

To deal with the above problem, it is necessary for the Sphere disk garbage collector to consult the mutator as to whether it can reclaim an object that appears unreachable. However, it is not trivial for the mutator to determine whether it still holds a reference to a given object. The scenario illustrated above is the simple case of the problem, as the reference field that is copied in object **C** has been swizzled before or during this operation[6]. This causes **B** to be added to the resident object table of the mutator (↝§2.4). If the garbage collector tries to reclaim **B** and consults the mutator, a look-up on this table will be sufficient to determine whether **B** can be reclaimed or not.

However, there are subtle cases when the contents of objects are copied *without* being swizzled. For example, in the case of PEVM, an invocation of the `clone()` method [GJS96] will result in the object contents to be copied with a memory copy operation and not in a field-by-field manner, therefore if reference fields contain PIDs these will not be swizzled [Lew00]. Therefore, in this case, a look-up on the resident object table *cannot* determine whether an object can be reclaimed from the store or not.

Currently, a satisfactory solution to this problem has not been finalised. It seems certain that the Sphere public API has to be augmented with an additional up-call to the mutator, through which the GC will consult the mutator whether it can reclaim any object it finds unreachable. This operation will take place sometime during the marking phase (↝§6.3.4.2) and the GC will have to treat as new roots every object the mutator still has a reference to. The action of the mutator to deal with this remains future work. Enforcing swizzling on all copied references or a variation on this is probably part of the solution.

6.6 The Sphere Global Garbage Collector

As a concurrent cyclic garbage collector has not yet been implemented in Sphere, it was necessary to provide a temporary solution for the Sphere users so that they are able to completely garbage collect their stores when they become full. To achieve this, an off-line stop-the-world disk garbage collector was developed. This is referred to as *Sphere Global Garbage Collector* (SGGC). There were two main requirements for it.

- ❑ To be fault-tolerant, unlike the off-line disk garbage collector of PJama Classic [Pri96, Ham97].

[6] It is common in persistent systems for a reference to be swizzled when copied from one object to another. For example, `a.field1 = b.field2;` in PEVM (↝§2.5) will cause `a.field1` to be first copied to the runtime stack, then copied in `b.field2`. As references are always pushed on the runtime stack swizzled [LM99, Lew00], `a.field1` will also be swizzled. In Napier88 [MCC+99], a similar invariant is maintained [CCM99].

- To maximise the re-use of Sphere's infrastructure, especially the already-existing compacting garbage collector, which was described in section 6.3.

To meet these requirements, SGGC operates in three phases.

❶ **Marking Phase** — First, it performs a marking traversal over the entire persistent store, starting from the persistent root and using a main-memory data structure for object-marking purposes. This data structure comprises a single array with one entry per partition in the store. Each array entry points to a bitmap that contains a number of bits equal to the maximum number of objects in a partition. Bitmaps are allocated lazily, upon the first access to the corresponding partition, and their size is determined by the size of the indirectory. Marking proceeds as follows: *(i)* find the appropriate bitmap, using the partition identifier extracted from the object's PID and *(ii)* set the corresponding bit, which can be calculated from the object's indirectory index, extracted again from the object's PID. The check as to whether the object is marked or not is performed similarly.

This phase performs no updates to the store so is by default fault-tolerant.

❷ **Nuking Phase** — At the end of the marking traversal, all unmarked objects are considered to be garbage, as they are not reachable from the persistent root[7]. All indirectories are then iterated over and any object that is not marked in the main-memory data structure is marked as garbage on disk; this will be referred to as the *"nuking"* of the object and is performed by setting its reference count field to its highest possible value, which will never be reached during normal operation (↝§4.5.3).

All nuking operations are logged to ensure that they can be undone or redone if a failure occurs. The entire marking phase takes place in terms of a single history, as it has to be atomic.

❸ **Sweeping Phase** — This reclaims the space taken up by garbage objects in the store by iterating over allocated partitions and invoking the Sphere compacting GC on them. The GC has been modified to deal with objects nuked by the marking phase and does not consider them roots into the partition.

All updates that take place in terms of the sweeping phase are performed by the GC. Since the GC is fault-tolerant, as described in section 6.3.1.2, the sweeping phase is also fault-tolerant.

Notice that, even though the operations above are fault-tolerant, the entire off-line garbage collection is also fault-tolerant but *not* atomic. This is the case because several histories are involved in a single garbage collection (one for the nuking phase and one per garbage collected partition) and once they are committed, they cannot be rolled back. Instead, if there is a failure, garbage collection has to be re-invoked until it completes successfully. It must emphasised though that, even if a failure prevented all garbage objects from being reclaimed, it will never leave the store in an inconsistent or unusable state.

A single utility that performs a full off-line garbage collection of a store, called SGGC, is distributed with Sphere. Additionally, two more utilities are also provided: *(i)* the *Sphere Global Marker* (SGM) that only performs the operation included in phases ❶ and ❷ above and *(ii)* the *Sphere Global Sweeper* (SGS) that only performs the operation included in phase ❷ above. These are useful on their own, i.e. an invocation of SGM will nuke all garbage objects, including cross-partition cycles, which can then be reclaimed by the concurrent partition GC. Additionally, SGM has been augmented to obtain statistics on the contents of the store.

Appendix D includes the user's guide for the SGGC, SGM, and SGS utilities.

[7] As SGGC operates off-line, there is no mutator to cache PIDs and introduce the complication described in section 6.5.

6.7 Garbage Collection as an Evolution Mechanism

One of the original requirements for Sphere was support for flexible *Schema Evolution* (↝§3.2). This involves changing the type of persistent objects (instances in the case of PJama) to reflect changes in the implementation of the persistent application. Evolution might only involve changes in the object representing a persistent class (imposed, for example, by the modification of a method) or changes in the layout of objects of the same type (imposed, for example, when a new field is added to a class).

Any evolution-related modifications need to be applied in a type-safe manner so that the consistency and referential integrity of the application are not jeopardised. For example, a method cannot be removed from a class while it can still be invoked or a field cannot be removed from an object while it can still be accessed. As Sphere only provides low-level storage facilities and cannot reason about the semantics of object graphs, ensuring type safety during evolution is performed at the mutator level.

Objects that are modified during evolution need to retain their identity (i.e. PID) since, even though some aspects of their implementations change, they are still considered to be the same objects and have to continue interacting with the objects they are reachable from. However, as layout changes might modify their size, any in-place transformations are not always possible as no padding is included between objects within a partition. Additionally, during evolution, both the old and new versions of an object that is being evolved should be available since the code that is performing the evolution might need to access the old state in order to produce the new state. This is the case because sometimes it is not enough to just set new fields to default values (e.g. might have to derive the values for new some fields from old ones).

The implementation of the evolution facilities of Sphere relies on augmenting the Sphere compacting GC with some new facilities. First, given a set of classes whose instances need to be evolved, partitions that contain such instances are identified. This can be done efficiently, by checking whether the corresponding descriptor has been allocated in the partition (↝§4.11.1). Then, using the compacting GC, the partition is moved to a new location and enough space is allocated in it for two images of each object that is being evolved: *(i)* the original one and *(ii)* the evolved one. At this point the indirectory entry still points to the original image, so that it can still be accessed normally, if necessary. By the end of the evolution, the new contents of the evolved objects have been installed over the evolved image. So far, the fault-tolerance of the GC also ensures the fault-tolerance of the evolution operation. If there is a crash during evolution, any evolved images that might be left inside partitions will simply be reclaimed during the next garbage collection. The operation commits by changing the indirectory entries of all processed objects to point from the original image of each object to the evolved one (all these updates must be logged to ensure the atomicity of this operation). The original images left in the partitions will be reclaimed during the next garbage collection.

A detailed description of the operation of schema evolution is given by Hamilton *et al.* [HAD99] and related high-level issues are discussed by Dmitriev and Atkinson [Dmi98, DA99b]. Unavoidably, all this work is targetted for the PJama system ($PJama_1$ in particular) because as mentioned above a lot of the operations (e.g. identification of instances) are PJama-specific and cannot be easily generalised. Naturally, all the related code is in the application-specific part of Sphere and outside its core.

6.8 Future Work

Even though a lot of thought and effort was put into the Sphere design and implementation to accommodate a concurrent disk garbage collector, the work on the garbage collector itself is still in its infancy and only an initial prototype has been implemented. The initial results are encouraging and prove the soundness of the framework (⇝§6.4), but some areas need improvement. As the GC elapsed times also affect the writer thread blocking times, the former have to be decreased for the GC's non-disruptiveness to be improved. Alternatively, finer-grain locking or other schemes to allow writer threads to update partitions during garbage collection might be explored.

Another way to decrease the GC elapsed times is to use large transfer units to install the partition contents in to-space. As described in section 5.3.5, this contributed to the decrease of object-promotion times by almost a factor of five and has the potential to decrease GC elapsed times too. Additionally, it might be possible to read small partitions entirely into memory, process them, and write them to disk with a single I/O operation. Apart from decreasing the number of I/O operations, this can also decrease the interference of the GC with the mutator threads at the disk-cache level as both the read operation to fetch from-space into memory and the write operation to install to-space can bypass the disk cache. This is safe as in the former case the partition cannot be updated since writer threads are blocked and in the latter case no other thread can access from-space until the partition move has been committed.

To achieve a good trade-off between GC disruptiveness and throughput, heuristics could be introduced to determine which partition and when to garbage collect. Such heuristics were pioneered by Cook *et al.* [CWZ94, CKWZ96], which report that they have obtained encouraging results using them. In fact, the framework for the former is already in place in the PT (⇝§4.8.1).

As mentioned earlier in this chapter, another aspect of Sphere that is not being taken advantage of yet is the ability to allow multiple garbage collection algorithms to operate over the same store, applied to different partitions. In particular, a non-relocating algorithm is envisaged to be applied to partitions containing large scalar objects (images, sound clips, etc.). Collecting such partitions can be very efficient as, since the objects contained in them have no references, no marking phase is necessary and the liveness of the objects can be determined solely from their reference counts. Not relocating such large objects during garbage collection but de-allocating them in-place should also decrease the GC elapsed time further.

Another area that needs to be explored is the interaction between disk garbage collector and the mutator heap in order to provide an efficient solution to the complication described in section 6.5. However, this will probably mainly affect the mutator and not Sphere itself.

Finally, the Sphere garbage collection scheme would not be complete if a concurrent cyclic collector is not introduced. This is essential if Sphere is to support long-lived applications. Work is currently under way on this and the Sphere Global Garbage Collector (⇝§6.6) was only implemented as a temporary solution.

6.9 Related Work

This section presents related work in the field of persistent and fault-tolerant garbage collection. Section 6.9.1 includes algorithms for small-scale stable heaps and section 6.9.2 includes ones designed to operate on large POSs.

6.9.1 Solutions for Stable Heaps

This section describes garbage collection algorithms for stable heaps, i.e. memory heaps that survive power-downs by having their contents replicated onto disk and failures by using recovery facilities (e.g. logging) to ensure fault-tolerance. All the techniques presented here assume heap sizes smaller than the store sizes envisaged for Sphere, residency of the entire heap in memory, fine-grain synchronisation between the collector and the mutator, and a physical addressing model (i.e. object references change when objects are moved).

Probably the first work on fault-tolerant, or atomic as it is referred to in the paper, garbage collection was reported by Kolodner *et al.* [KLW89, Kol87]. It was implemented in the context of *Argus* [Lis84], a language for reliable distributed computing. It uses an adaptation of Cheney's two-space copying algorithm [Che70] to perform the garbage collection. As the algorithm modifies from-space to install forwarding pointers into to-space, the field of the object that was overwritten in from-space is written to the log so that it can be recreated following a failure. The algorithm operates in a stop-the-world-and-do-everything fashion.

Later, Kolodner *et al.* proposed an incremental version of the above algorithm [KW93], based on Baker's incremental two-space scheme [Bak78] (which in itself is an incremental version of Cheney's algorithm). It was targetted for large stable heaps and introduced incrementality in order to decrease the garbage collector-imposed pause times. Its operation is similar to Baker's algorithm, apart from the logging of updates in from-space (as described above) and some other recovery issues [KW93]. The synchronisation with the mutator is achieved by the method proposed by Ellis *et al.* [ELA88]. This takes advantage of virtual memory protection facilities, protects pages that contain newly-copied objects in to-space and, when one of them is accessed, copies all the objects reachable from that page into to-space and unprotects it. This ensures that the mutator never accesses references into from-space. Additionally, when the mutator applies updates to the heap, it has to log them for fault-tolerance purposes. Such log records are also available to the garbage collector so that it can discover new potential roots into from-space.

Detlefs' garbage collection algorithm [Det90, Det91], implemented in the context of the Camelot/Avalon system [EMS91], is very similar to Kolodner's incremental one in that it is also based on Baker's algorithm and uses the Ellis *et al.* synchronisation scheme described above [ELA88]. However, the main difference is that it does not have to generate log records when forwarding references are installed in from-space. Instead, a clever combination of page-pinning and ordering of page writes ensures that, if an object in from-space containing a forwarding reference is written to disk, its corresponding image in to-space is also guaranteed to have been propagated to disk.

A different approach was taken by Nettles and O'Toole [ON93, NOPH93, NOG93, ON94]. They have also proposed an incremental two-space algorithm. However, in this case, the mutator is accessing and manipulating the objects in from-space, rather than in to-space that Baker's algorithm assumes. The garbage collector, running as a separate thread, evacuates objects to to-space concurrently with the mutator's operation. It also reads the log, where any updates to objects in from-space have been noted, in order to re-apply them to to-space if necessary. This is referred to as *replication-based* or *replicating* garbage collection. Its main strength is that the only interaction between mutator and garbage collector is through the log and no explicit read-barrier has to be imposed on the mutator. A simplified version of this scheme was adopted for the Sphere compacting garbage collector (⇝§6.3).

6.9.2 Solutions for Large Stores

This section covers algorithms that are targetted for large POSs. All but one are based on a partitioned approach, i.e. the division of the store into smaller units called partitions so that each such unit is garbage collected independently. This approach was originally proposed by Bishop [Bis77]. Additionally, Yong *et al.* ocompared incremental copying, reference counting, and partitioned collection in a client-server environment and concluded that the latter performed the best [YNY94]. In some systems partitions are also referred to as segments [FS95] or cars [MMH96].

The only algorithm presented here that has not adopted a partitioned scheme is the one proposed by Skubiszewski *et al.* [SV97, SP97, SP96]. This introduces a novel method of synchronisation between the store and the mutator. A *GC-consistent cut* is a set of virtual copies of store pages, taken in such a way that if an object is unreachable in the store, it is also unreachable in the cut. The garbage collector examines these copies, rather than the store pages, to determine whether objects are garbage. The copies are created from information contained in the log, hence no further interaction with the mutator is needed. This algorithm has been implemented in the context of the O_2 system [BDK92] and it is believed it is the only one with published measurements using stores that exceed a gigabyte in size. It requires a single global marking phase over the entire store and uses main-memory data structures for marking purposes [Sku97]. This has the disadvantages that *(i)* even though it operates concurrently, it takes literally hours to complete a garbage collection over a store of size of a few gigabytes [Sku97], *(ii)* if a crash occurs the entire garbage collection operation will be discarded, and *(iii)* the size of the main-memory data structures is linearly proportional to the store size and grows to several tens of megabytes for stores of a few gigabytes [Sku97].

Moving to partitioned schemes, Amsaleg *et al.* have proposed a concurrent algorithm for a client-server database system [AFG94, AFG95, AFG99]. It has been implemented in the context of the EXODUS system [CDRS86, CD87] and its design goals include minimal log traffic, non-disruptiveness, efficiency with respect to I/O, and holding no locks. The implementation is based on a traditional mark&sweep collector [Jon96] that has been augmented with a set of invariants to ensure safety and fault-tolerance. In particular, all objects that participate in a transaction are linked in a linked list until the end of the transaction. This ensures that none of them have been reclaimed if a transaction aborts and garbage objects become reachable again (↝§6.1.2).

A variant of the popular *Mature Object Space* (MOS) algorithm (also known as the *Train* algorithm) [HM92, SG95] has been proposed by Moss *et al.* [MMH96] It is referred to as *Persistent MOS* (PMOS). Like MOS it assumes that the store is split into a number of *cars* and cars are linked into *trains*. Remembered sets of references in and out of cars are maintained in the store and in main-memory respectively. The latter are calculated every time a page is fetched from and evicted back to disk, potentially imposing a performance overhead. The garbage collector can operate over a few cars at a time and evacuates the reachable objects to other cars where there are references to them. This is done in a way that all garbage cycles will eventually be evacuated to the same car or train and hence can be collected.

An initial and simple implementation of PMOS has also been reported [MBMM98]. However, even though PMOS guarantees that all garbage cycles will eventually be reclaimed, the space requirements for the remembered sets can be excessive, if complex object graphs are used. This is an inherent drawback of the MOS algorithm, already experienced in an implementation of it [Gar99], and could be even more drastic in the context of a very large POS. A variant of PMOS targetted for distributed systems, called *Distributed MOS* (DMOS), has also been reported [HMMM97]. However, as it assumes no failures, it seems more

appropriate for large closely-located computer clusters instead.

The garbage collection work in the Thor system [LAC+96], reported by Maheshwari and Liskov [ML97, Mah97], assumes a framework similar to Sphere. A partitioned approach has been adopted and lists of references both into and out of each partition are maintained. The latter are in fact only needed to allow efficient manipulation of the former. Two garbage collectors are employed: *(i)* one that collects a single partition using the in-lists as roots and *(ii)* one that performs a global marking phase in order to identify cyclic garbage. The latter is piggy-backed on the former. Additionally, care has been taken to minimise I/O accesses, especially when maintaining the in- and out-lists, by caching the deltas over them and applying them lazily. Identification of garbage cycles in subsets of the store is not supported.

A different approach is taken by the garbage collection algorithm adopted in the Larchant [SF95, FS95, FS96, BF98]. Replicas of data segments are cached at each client. Each data segment includes in- and out-lists. As several segments are cached at a client, a local marking phase over these segments can identify garbage cycles that span over them. However, garbage cycles can only be collected if the segments that contain them happen to be cached in the same client.

The GemStone/J product from GemStone Inc. [Gem98b, Gem98a] also relies on garbage collection for the reclamation of unreachable objects. In fact GemStone are possibly the only commercial organisation that supports this approach over the explicit de-allocation alternative [And98]. Unfortunately, details about the implementation of their system are not disclosed. It seems however that their collector is epoch-based. Newly-allocated objects might be collected by a concurrent collector. However longer-lived ones can only be collected by a batch off-line utility [Gem98b].

Work by Cook *et al.* concentrated on policies on which partition to garbage collect [CWZ94] and when to initiate a garbage collection [CKWZ96] in order to achieve a good balance between garbage collector throughput and disruptiveness. They claim that such decisions can only be taken dynamically and depend heavily on the behaviour of the application.

Finally, what is unfortunately *missing* from the literature is a standard benchmark to evaluate disk garbage collectors. This was identified by Amsaleg *et al.* [AFG95] and also by Maheshwari and Liskov [ML97]. Even though several benchmarks exist to evaluate the performance of POSs, like the infamous OO7 [CDN93], they are not entirely appropriate for garbage collection work and currently ad-hoc solutions are usually employed. The introduction of such a benchmark would greatly benefit future research into this area.

The difficulty in defining an effective disk garbage collection benchmark, as opposed to one targetted for main memory, mainly lies in the differences in requirements between the two approaches. In particular, as POSs are typically much larger and longer-lived than main-memory heaps, any brief microbenchmarks will not provide realistic results (even though such benchmarks are routinely used to evaluate main-memory collectors [Zor92, SP93, PD00]). Instead, a long-lived benchmark with several mutator threads (or clients in a client/server environment) providing a realistic "real-world" load should provide more useful results. Unfortunately, such benchmarks are difficult to arrange in practice (due to potential machine failures, interference from late-night backup activity, etc.) and especially repeat so that different approaches can be fairly compared.

6.10 Summary

This chapter covered garbage collection issues in Sphere. It discussed the need for disk garbage collection in POSs (↝§6.1) and the framework set up in Sphere to support it (↝§6.2). It gave a detailed description of the Sphere concurrent compacting garbage collector (↝§6.3) and evaluated the first prototype of it, which proved the feasibility and soundness of the design (↝§6.4). The necessary interaction between the disk garbage collector and the mutator was discussed (↝§6.5), followed by the global collector, introduced as a temporary measure until a concurrent cyclic one has been implemented (↝§6.6), and how the compacting collector was modified to also support schema evolution (↝§6.7). Then the future work was presented (↝§6.9), followed by related work (↝§6.8).

The next chapter presents performance and ease-of-use evaluation of the PJama system running over Sphere against two other persistence mechanisms.

She'll make point five past lightspeed. She may not look like much but she's got it where it counts, kid. I've made a lot of special modifications myself.
— **Han Solo**, *Correlian Smuggler*

Chapter 7

Performance Evaluation

This chapter presents the experiments that were undertaken to compare the performance of the PJama system (⇝§2) running over Sphere (⇝§3 and §4) against two alternative systems that provide persistence for Java: a commercially-available persistent engine and the standard Java Object Serialisation facility (⇝§2.3.4). Section 7.1 describes the above three systems and section 7.2 presents the benchmarks that provided the load for the performance experiments. Section 7.3 concentrates on ease-of-use differences and facilities that the three persistence mechanisms provide and section 7.4 focuses on the performance evaluation results. Then, section 7.5 summarises the results and section 7.6 concludes the chapter.

7.1 Platforms Compared

The three systems compared in this section are the following.

❶ **PJama Version 1.6.6** — The latest implementation of the PJama system, described in section 2.5. It relies on Sphere release 1.3.7.2 for its storage requirements.

❷ **Java Object Serialisation (JOS)** — A standard facility written in Java that can serialise/deserialise object graphs to/from a bytestream, which can then be transmitted over the network or, in the case of persistence, stored on disk [Sun98b]. It is described further in section 2.7. Only instances of classes that implement the `java.io.Serializable` interface are allowed to be serialised. JOS is part of the Java core API [Sun98b, Sun98c] and is considered to be the default persistence mechanism of the Java language. However, its simple, non-incremental, and non-transactional approach to persistence has a number of disadvantages that are enumerated in section 2.7 and discussed further by Evans [Eva00] and D. Jordan [Jor99a].

❸ **A Commercially-Available Persistent Engine (CAPE)**[1] — A commercial persistent object store entirely written in Java. It is a single-user system that supports multiple concurrent transactions and

[1] Unfortunately, due to restrictions in the licensing agreement, the name of this product cannot be disclosed. It will be referred to as CAPE throughout this chapter.

incremental object faulting and updating. For instances of a class to be allowed to persist, its `.class` file has to be annotated with CAPE-specific code that adds residency checks (⇝§2.4), dirty-object tracking, etc. The annotations are applied by a postprocessor that is distributed with the system. Annotated classes are referred to as *persistence capable* and cannot be used in non-CAPE applications, because of the bytecode modifications.

To make the comparison between the three systems above as fair as possible, JOS and CAPE were executed by the same JVM that was modified to produce the above PJama release, namely Sun JDK for Solaris, production release, version 1.2.2_05a; this includes the same memory management, garbage collector configuration, JIT compiler, etc.

All experiments presented in this chapter were run on the machine described in section 4.3. However, since it was not possible for JOS and CAPE to use raw disk partitions, all three systems used Unix files instead. In an attempt to flush the operating system disk buffers and obtain more reliable cold results, a large file was serially read between runs. All times given in this section were taken with the `System.currentTimeMillis` method [Sun98c] and are the average of ten runs, with the worst one removed. Finally, the heap size was fixed to a value large enough so that all persistent objects, which needed to be accessed, could reside in memory and no object-eviction was performed. It is worth noting however that for most of the benchmarks presented below, CAPE needed around twice as much memory as the other two systems. This extra overhead was most likely to be due to the large data structure that CAPE needs to map persistent identifiers to memory-resident objects (the equivalent of the resident object table of PJama, ⇝§2.4).

7.2 Benchmark Description

The benchmark that provided the load for the experiments presented in this chapter was the manipulation (creation, querying, and update) of a simple binary tree. Each tree node contains a key (a `java.lang.String` object), according to which the tree is ordered, a value (an `int`), and two references to its children. Unique randomly-generated 20-character strings were used for keys and random integers for values. The tree-manipulating code was written entirely in Java and is shared by the three persistence mechanisms, after being annotated appropriately when necessary (e.g. postprocessed for CAPE).

A single experimental run involved populating the tree with a number of nodes and stabilising it to a store. All the trees that were constructed with the same number of nodes were identical, because the keys were read from a number of already-generated files. Once the store was populated, two read-only programs were executed against it (a complete tree traversal and a series of look-ups on the tree), followed by a read-write program (updating the values of some of the tree nodes and committing the updates to the store). The `main` method of each of the above programs was written specifically for one of the persistence mechanisms, however they all shared the tree-manipulating code.

Stores of variable sizes were created by varying the number of nodes inserted into the tree; between 200,000 and 2,000,000 nodes were used in the experiments described below. Because each tree node comprises three objects (the node object itself and the two objects that make up the Java string representing the key [Sun98c]), theg enerated stores contained between 600,000 and 6,000,000 objects.

Finally, it must be emphasised that this benchmark is totally artificial and was developed exclusively for the purpose of obtaining the results presented below. Still, its synthetic nature allowed the variation of several

	JOS	CAPE	PJama
Imposed Changes			
JVM Changes			yes
Source Changes	yes		
Bytecode Changes	yes	yes	
Extra Lines of Code*	94	103	57
Facilities			
Incremental Object Fetches		yes	yes
Incremental Object Updates		yes	yes
Maintains Object Identity		yes	yes
Fault-Tolerance		yes	yes
Transactions		multiple/concurrent	single/global†
Persistence Model			
Orthogonal Persistence			yes‡
Persistence By Reachability	yes	yes	yes
Summary			
Ease of Use	easy	medium	easy

* The number of lines of code, specific to each persistence mechanism, that was necessary to be written in order to implement the benchmarks presented in section 7.2. This does not include the shared classes and non-essential information (comments, blank lines, output statements, etc.).

† Even though the PJama system at the moment only supports a single global transaction, work is under way to augment it with concurrent multiple transactions [Day96, DAV97].

‡ See section 2.3 for some exceptions to this claim.

Table 7.1: Evaluation — Facilities / Ease-of-Use Comparisons

parameters (total size of the tree, number of fetched objects, number of updated objectsm, etc.) in order to test specific aspects of the three systems that were compared.

7.3 Facilities / Ease-of-Use Comparisons

Table 7.1 summarises the facilities and ease-of-use aspects of the three persistence mechanisms, which were used to obtain the performance results presented later in this chapter. As shown in this table, PJama is the only system that requires a specialised JVM, whereas that others can run with a standard one. However, PJama allows classes to persist totally unchanged, whereas CAPE requires changes to the bytecodes, as described in section 7.1, and JOS potentially requires changes to the source and hence bytecodes also[2]. Further, the minimum number of lines of code that needed to be written to implement the benchmarks described in section 7.2 for PJama was 57, compared with 94 and 103 for JOS and CAPE respectively. The reason for this is that the PJama API (presented in section 2.3.2) is truly minimal, compared to the CAPE API that requires more method invocations in order to achieve similar results. Additionally, most of the extra code in the case of JOS deals with manipulating the file that contains the bytestream.

[2]The required change is minimal, i.e. the programmer has to arrange so that every class that needs to be serialised implements the `java.io.Serializable` interface [Sun98b]. However, this requirement can prevent classes obtained in bytecode-only format from being used by JOS, if they do not already implement this interface.

	Operation	Store Access	Tree Nodes Visited*
§7.4.1	Store Creation	write-only	100%
§7.4.2	Tree Traversal	read-only	100%
§7.4.3	Key Look-Ups	read-only	around 13%
§7.4.4	Value Updates	read-write	5%

* The percentage of individual tree nodes that each operation visited.

Table 7.2: Evaluation — Summary of the Experiments

Table 7.1 also illustrates the number of facilities that each of the persistent systems provides. Both PJama and CAPE provide incremental object fetches and updates, whereas JOS always has to deserialise/serialise all the objects of the object graph in order for any of them to be accessed. PJama and CAPE also maintain the identity of an object across JVM invocations, while JOS essentially copies an object's contents into the bytestream, hence the same object can be serialised into several bytestreams or deserialised by several JVMs or even within the same JVM. Additionally, CAPE provides concurrent multiple transactions and PJama currently only provides a single global transaction. JOS does not need to provide such facilities as it never updates already-existing bytestreams and always creates new copies of them.

Considering the persistence model of the three systems, PJama is the only one that can be referred to as orthogonal, as both CAPE and JOS require bytecode and source changes respectively in order to store instances of some classes (↝§2.3). All three systems embrace persistence by reachability, even though this is limited in the case of CAPE and JOS, as this only applies to objects that are instances of persistence capable classes.

Finally, JOS and PJama are equally easy to use, in the former case because of its simplistic approach and lack of sophisticated facilities and, in the latter case, because of the power of its persistence model. Even though not by any means difficult, CAPE is slightly more complex to use as its API is not as minimal as that of PJama and JOS and a programmer has to postprocess some of the `.class` files before they can be used by CAPE. However, the biggest nuisance of CAPE is that, after a transaction is committed and the corresponding updates have been propagated to the store, the in-memory image of all updated objects becomes "hollow" and these objects have to be refetched from the store.

7.4 Performance Results

This section presents the performance results of the experiments. Table 7.2 provides a summary of the experiments and provides an index to the corresponding sections.

7.4.1 Store Creation

> *"Populate the tree in main memory and time how long it takes to stabilise it to a newly-created persistent store."*

The purpose of this benchmark is to measure the bulk object-loading facilities of the three persistence mechanisms.

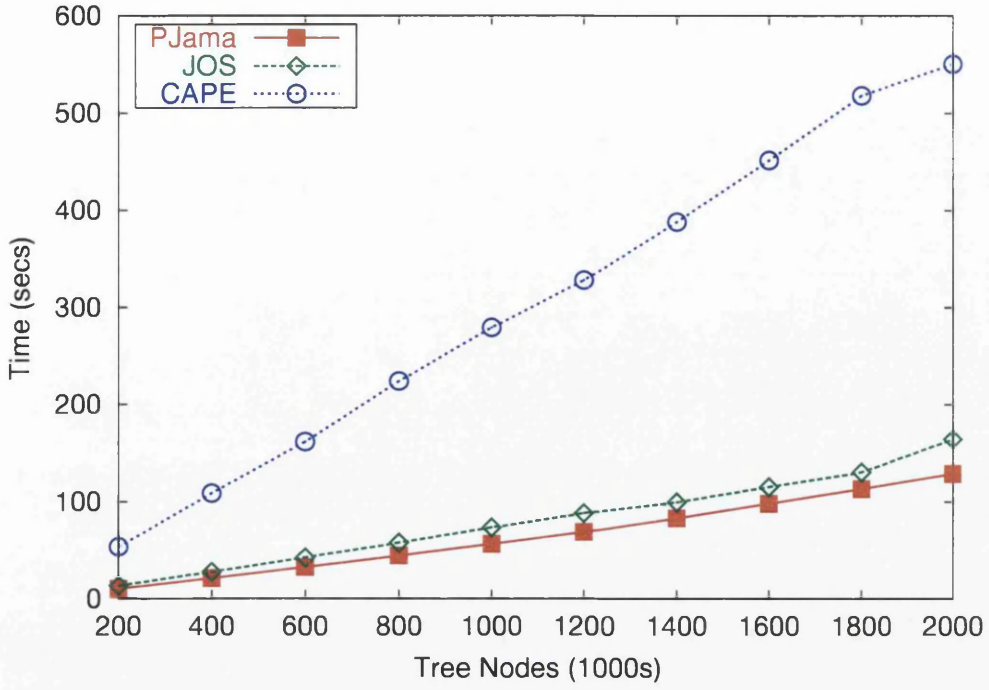


Figure 7.1: Evaluation — Store Creation

Figure 7.1 plots the timing results of stabilising between 200,000 and 2,000,000 tree nodes (this corresponds to 600,000 to 6,000,000 objects, as described in section 7.2). The results were obtained by timing the `OPRuntime.checkpoint` method in the case of PJama [Sun98f], the `ObjectOutputStream.writeObject` method in the case of JOS [Sun98b, Sun98c], and the corresponding transaction-commit method in the case of CAPE. The times do not include the initial population of the tree in main memory. The figure shows that PJama is fastest, with JOS performing around 22% slower. CAPE is the slowest of the three, being around 4.2 times slower than PJama. It is also worth noting that the PJama system, unlike the other two, stabilises a number of additional bootstrap objects and classes upon store creation.

7.4.2 Tree Traversal

"Perform a depth-first traversal of the tree, visiting all node objects."

The purpose of this benchmark is to measure the performance of the three systems when a large percentage of persistent objects are visited (all nodes of the tree are visited exactly once).

Figure 7.2 illustrates the timings when starting with a cold heap, i.e. when having to fault-in all visited persistent objects. In the case of PJama and CAPE, the results were obtained by timing the tree-traversal method during which their residency checks faulted-in all necessary objects. In the case of JOS, the results were obtained by timing the combination of the `ObjectInputStream.readObject` method that deserialised the bytestream [Sun98b, Sun98c] and the tree-traversal method. The figure shows that PJama is again fastest, with CAPE performing 4.6 times slower and JOS performing 5.2 times slower.

A comparison between figures 7.1 and 7.2 surprisingly reveals that JOS performed considerably faster when serialising than when deserialising the bytestream. Additionally, it is worth pointing out that JOS had the

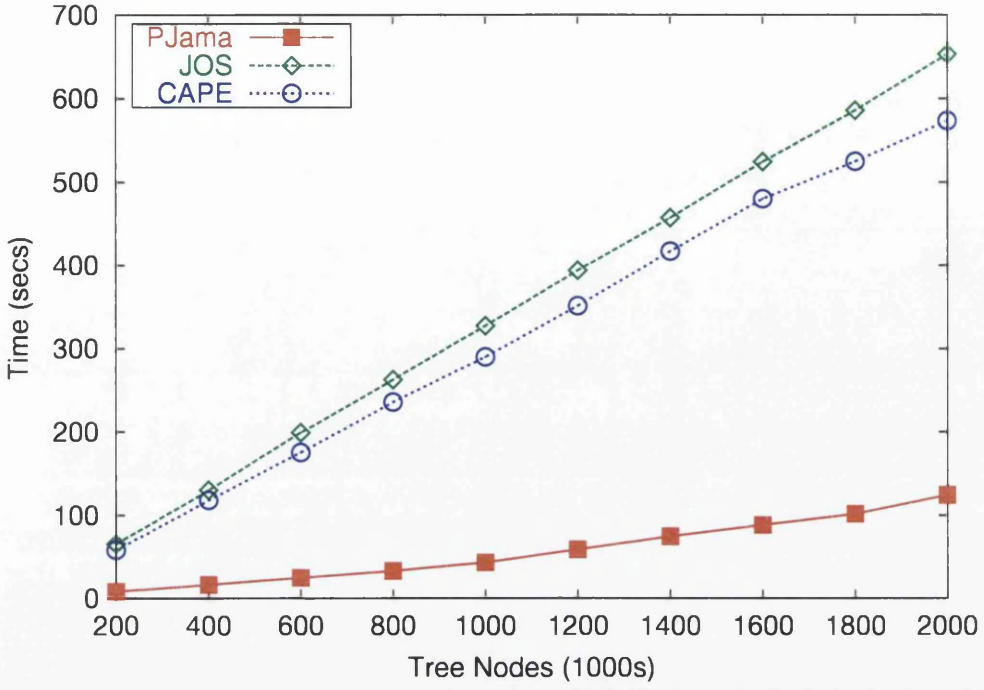


Figure 7.2: Evaluation — Tree Traversal (Cold)

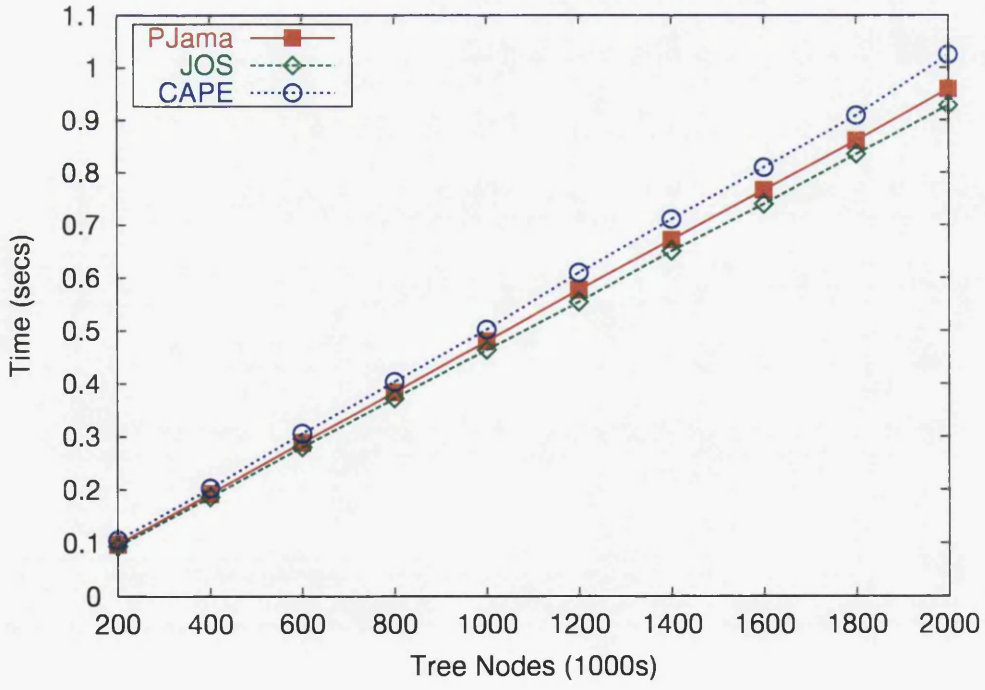


Figure 7.3: Evaluation — Tree Traversal (Hot)

overhead of having to deserialise the entire bytestream before the traversal could commence. It turns out that more than 99% of the JOS timings, presented in figure 7.2, was the deserialisation process and less then 1% the actual tree-traversal; this is not illustrated explicitly in the figure. On the other hand, PJama and CAPE perform on-demand object-faulting, therefore their responsiveness is immediate.

Figure 7.3 plots the traversal times, when starting with a hot heap, i.e. all the persistent objects have already been faulted into main memory and no disk accesses take place during the traversal. In this case JOS performs best and is marginally faster than PJama. The reason for this is that PJama still has the overhead of the residency checks planted in the JVM and the JIT-compiled bytecodes [LM99], whereas JOS does not. In turn, PJama is slightly faster than CAPE; this is because CAPE also has the overhead of residency checks, however these are performed at the bytecode-level and are slower than those of PJama that are performed at the JVM-level.

7.4.3 Key Look-Ups

"Perform a series of key look-ups, only visiting a small subset of tree nodes."

The purpose of this benchmark is to measure the performance of the three systems when a small percentage of persistent objects are visited, with some of them being visited multiple times.

Figure 7.4 plots the timing results of the look-ups for the three systems, when starting with a cold heap. The number of look-ups performed each time was 2% of the number of tree nodes, with 50% of them being successful and the other 50% failing. For every different tree size, the keys that were looked up were exactly the same and read from the same file. However, the keys were read into memory before the benchmark started so that the corresponding disk accesses did not skew the timing results. The timings were obtained in a manner similar to section 7.4.2. The figure shows that PJama is fastest, with CAPE being 31% slower. However, JOS suffers from the lack of incremental object-fetching facilities and needs to deserialise the entire bytestream before it can perform the look-ups, hence it is 6 times slower than PJama. The lack-of-responsiveness issues mentioned in section 7.4.2 apply in this case too.

Figure 7.5 plots the look-up times, when starting with a hot heap. The results follow the trend of figure 7.2, with JOS being marginally faster than PJama and PJama being slightly faster than CAPE (the reasons for this are explained in section 7.4.2).

7.4.4 Value Updates

"Increment the values of a small percentage of tree nodes and commit the updates to the store."

The purpose of this benchmark is to measure the performance of the three systems when a small percentage of objects are dirtied and the updates propagated to the store.

Figure 7.6 plots the timing results of the value updates, after they were performed in main memory, when starting with a cold heap. Only 5% of the tree nodes were visited and dirtied. The times were obtained in a manner similar to section 7.4.2 and do not include the propagation of the updates to the store. The figure shows that, once more, PJama considerably outperforms the other two systems. Again, the lack of responsiveness and facilities for incremental object-fetching in JOS, mentioned in sections 7.4.2 and 7.4.3 respectively, apply in this case too.

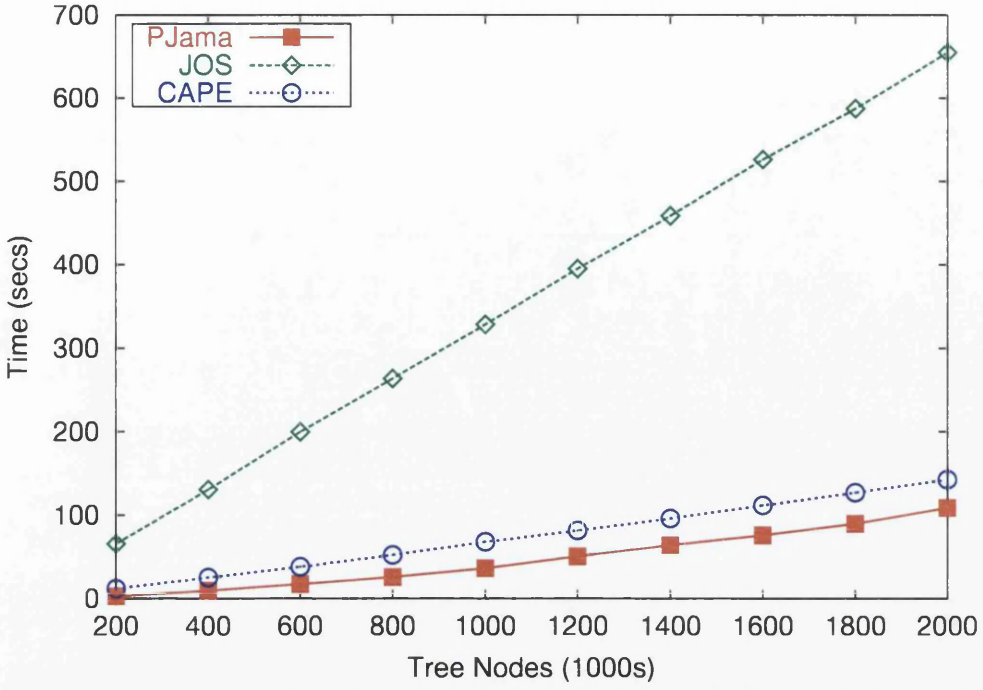


Figure 7.4: Evaluation — Key Look-Ups (Cold)

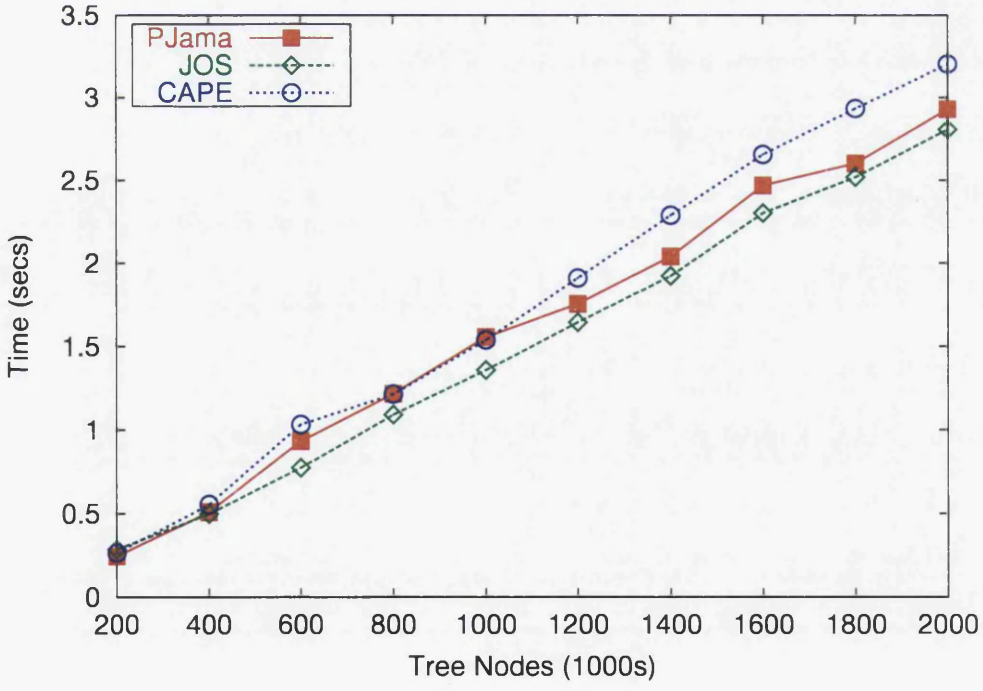


Figure 7.5: Evaluation — Key Look-Ups (Hot)

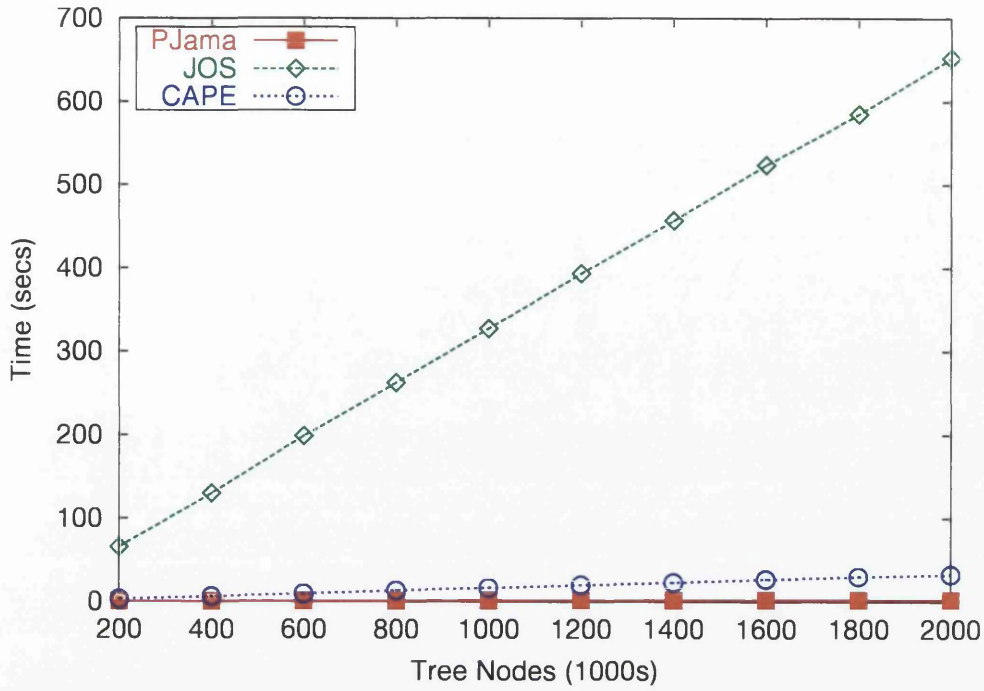


Figure 7.6: Evaluation — Value Updates (Cold)

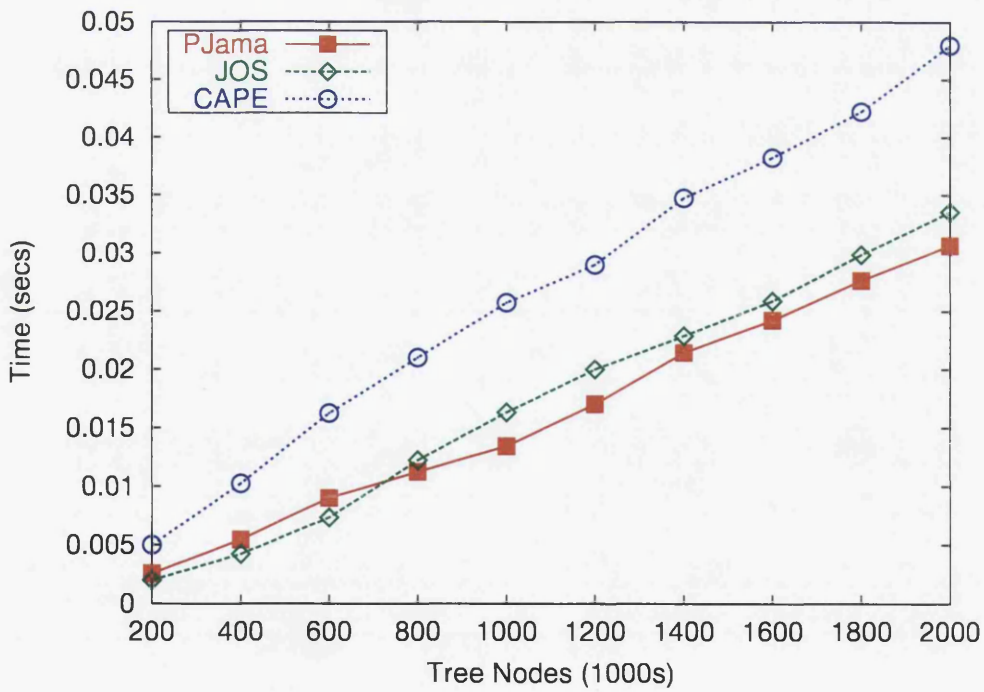


Figure 7.7: Evaluation — Value Updates (Hot)

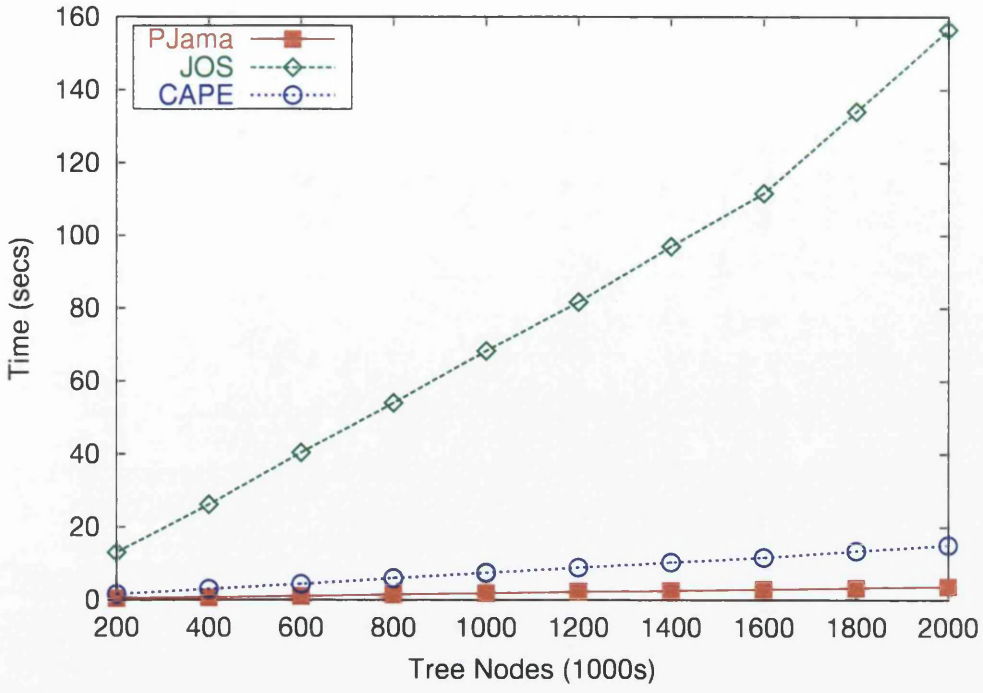


Figure 7.8: Evaluation — Value Updates (Commit)

Figure 7.7 plots the main-memory update times, when starting with a hot heap; again these exclude the propagation of the updates to the store. The figure shows that, unexpectedly, PJama performs best, operating marginally faster than JOS. This is a surprising result and not consistent with the hot traversal and look-up timings illustrated in figures 7.3 and 7.5. This inconsistent behaviour could be a result of experimental error, as the times in question are relatively small (up to 33ms) and the granularity of the `System.currentTimeMillis` method might not be accurate enough. Finally, CAPE performed 57% slower than PJama.

Figure 7.8 plots the timing results of propagating the above updates to the store. The results were obtained in a similar manner to section 7.4.1, by timing the `OPRuntime.checkpoint` method in the case of PJama [Sun98f], the `ObjectOutputStream.writeObject` method in the case of JOS [Sun98b, Sun98c], and the corresponding transaction-commit method in the case of CAPE. The figure shows that PJama performs best, with CAPE being 4.1 times slower (which is similar to the ratio of the store creation times, presented in section 7.4.1). Finally, JOS suffers from the lack of facilities to update individual objects and is around 43 times slower than PJama, as it has to serialise the entire object graph even though only a small proportion of the objects had actually been updated.

7.5 Discussion

The results presented in section 7.4 illustrate that PJama running over Sphere consistently outperforms the other two systems. The only time when it does not do so is during hot runs, where the residency checks planted at the JVM impose a marginal performance penalty compared to a "vanilla" JVM. However, this

penalty is extremely small[3] (only around 3% in the case of the hot tree traversals, ⇝§7.4.2) and the residency checks at the JVM-level allow PJama to outperform CAPE's incremental object-fetching facilities, that rely on residency checks planted at the bytecode-level. Additionally, the ability of PJama and CAPE to operate on individual objects allows them to scale better and provide better overall performance and responsiveness than JOS, in the case of all cold runs that visited a small proportion of the tree nodes (⇝§7.4.3 and §7.4.4).

So far, this thesis has claimed that the persistence model and implementation approach adopted for the PJama system can provide superior performance and ease of use compared to alternative systems. The above discussion, based on the experimental evidence presented in this chapter, as well as the facilities and ease-of-use comparisons presented in section 7.3, re-enforce this claim. Even though the behaviour of the benchmark used to obtain the timing measurements is synthetic and possibly atypical, similar performance gains have been experienced in practice with larger applications.

7.6 Summary

This chapter presented performance measurements which show that the PJama system has superior performance and responsiveness when compared to two alternative persistence mechanisms. These three mechanisms were first described (⇝§7.1), followed by an outline of the benchmarks that were used to obtain the performance measurements (⇝§7.2). Then, a comparison between the facilities and ease-of-use issues of the three mechanisms was given (⇝§7.3). It was followed by the timing measurements themselves (⇝§7.4) and a discussion on the obtained results (⇝§7.5).

The next chapter concludes the thesis and presents future work and possible directions.

[3]Research by Hosking *et al.* suggests ways of substantially reducing this overhead [HNCB98, BNHC98].

I've been waiting for you, Obi-Wan. We meet again, at last. The circle is now complete. When I left you, I was but the learner. Now, I am the master.
— **Darth Vader**, *Dark Lord of the Sith*

Chapter 8

Conclusions and Future Work

This thesis presented a detailed overview of Sphere, a new persistent object store from the Department of Computing Science at the University of Glasgow, Scotland. It first gave an overview of persistent object stores (⇝§1) and the context in which Sphere was developed (⇝§2). The high-level design and low-level implementation details followed (⇝§3 and §4). Then, the thesis concentrated on the bulk object-loading *Ghosted Allocation* algorithm (⇝§5) and Sphere's disk garbage collection framework (⇝§6). Finally, experiment results illustrated the performance and ease-of-use advantages of using PJama and Sphere over two alternative persistence mechanisms (⇝§7).

Sphere has been operational for about a year and supports PJama_1, the latest implementation of the PJama system (⇝§2.5). This has been publically released since the beginning of December 1999 [Thewww]. The integration of Sphere with the VM was trouble-free, proving Sphere's ease-of-use. This is strengthened by the fact that the team that performed the integration was *not* the Sphere development team. Its adaptability and the flexibility of its architecture was proved by the fact that, even though the implemented object kinds and partition regimes were originally targetted for the Classic JDK VM (⇝§2.4), they were adopted in the PJama_1 VM (⇝§2.5) mostly unchanged. Additionally, a new member of the development team implemented a new object kind in under a day, even though at the time they were unfamiliar with the Sphere code base.

Sphere has sustained several applications that operate on top of the PJama_1 system. These most notably include a GIS system that has imported the TIGER/Line data [US 98, US www] for the state of California (over 3GB store) and the *Portable Business Object Benchmark* (pBOB) [BDF+00] (4.9GB store, 24 warehouses, each with 5 threads). The largest single object Sphere has been known to have stored was 34MB (a single scalar array). More such applications are expected to be developed in the near future. In addition, speed improvements between 6 and 16 times have been reported when using PJama_1 rather than PJama Classic. This has been mainly due to the introduction of the JIT and the much more advanced memory management of PJama_1. However, it also indicates that the more complex and heavy-weight architecture of Sphere, compared to the simple POS of PJama Classic, does not impose a critical performance hit on the overall performance of the system, while providing the advantages discussed throughout this thesis. Finally,

Sphere has proved robust and reliable, with only about half a dozen bug reports having been filed.

In addition, as it was developed in an academic environment and is part of a research-oriented project, Sphere was also designed to be used as a research platform. So far it has performed this duty well, providing a stable base for the promotion experiments described in chapter 5, the initial garbage collection work described in chapter 6, and the initial schema evolution facilities discussed elsewhere [HAD99].

Future work on Sphere will concentrate on the following areas.

- **Garbage Collection** — Unfortunately this is currently the least developed aspect of Sphere, due to the fact that initial effort was put into other areas (mainly robustness, bulk object-loading, and schema evolution), as these were absolutely necessary in order for $PJama_1$ to be used at all. Future work, as discussed in section 6.8, will be focused on increasing garbage collection throughput by using heuristics on when and which partition to process, decreasing elapsed times (which directly affect the blocking times of writer threads), minimising its disruptiveness on the rest of the system, and most importantly introducing a cross-partition garbage cycle removal mechanism.

- **Performance** — Even though the current performance of Sphere is very good, there is still room for improvement. Using larger transfer units (possibly of different sizes for different parts of the store, i.e. object space, indirectories, partition table) is one area that needs to be explored. This provided the largest performance improvement in the promotion experiments (⇝§5.3.9) and there is evidence that it can also improve object-faulting times as well. Other areas include using more asynchronous I/O operations, taking advantage of multiple physical disks, performing page prefetching, introducing object-clustering in an attempt to reduce I/O, and experimenting with partition regimes optimised for very large objects. Finally, a lot of parameter-tuning using large-scale long-running "real-world" benchmarks is necessary to identify and eliminate potential performance and scalability bottlenecks.

- **Adaptability** — A number of store parameters can be currently set in the Sphere configuration file (⇝§C). The number of these is likely to increase as further tuning and evaluation takes place. However, large systems with large numbers of parameters tend to be awkward to install and use to their full potential (an example of this is the Solaris file system [Ber98]). This is especially the case for naïve users who are not, and in most cases really does not have to be, familiar with the internals of the system's implementation. A notorious example of this is the Oracle database that sometimes needs a specialised engineer to even get it working at all, as experiences in the University of Glasgow have revealed [Atk99]. Furthermore, tuning parameters for one application will not necessarily provide maximum performance for others. It follows that dynamic tuning of parameters by Sphere itself, based on the applications running against it, is an important area that needs to be investigated.

- **Longevity** — As discussed in section 1.1, POSs are only useful if they can retain data reliably and safely for very long periods of time. Obvious facilities that contribute to this are fault-tolerance, robustness, provision for archiving, and automated free-space reclamation. These are currently operational in Sphere and are being improved upon. However, given the dynamic and ever-changing nature of data, schema evolution is a very important factor that contributes to the longevity of a store. The current approach was outlined in section 6.7. Future work will concentrate on improving it, making it more flexible and intuitive for the programmer, and exploring the possibility of performing schema evolutions incrementally.

In the future, larger and more complicated collections of data will be generated and/or collected and will need to be stored, queried, processed, and distributed. The following extract from a CNN news report gives a very recent example of this trend.

> *"In nine days and six hours of mapping, the astronauts surveyed 43.5 million square miles (111 million square kilometers) of the Earth's terrain at least twice. Double imaging is needed to create ultraprecise 3-D maps of the planet's peaks and valleys, as far north as Alaska and as far south as the tip of South America.*
>
> *The mapping gathered enough geographic data to fill 20,600 compact discs.*
>
> *It will take scientists one to two years to go through all the material the shuttle will bring back."*
>
> — from CNN on the completion of Shuttle mission STS-99 [NASwww], February 2000 [CNNwww]

According to the above, the Shuttle in nine and a half days gathered data that needs 13TB of storage space just to be stored, plus considerably more in order to be indexed. As technology progresses, storage requirements like this will become common. In order for them to be met, more sophisticated and scalable storage technology will be necessary. Even though Sphere cannot yet accommodate amounts of data that large, it has proven itself to be a robust and efficient storage technology, it has met its design goals and, most importantly, it provides a stable and flexible platform for future research in the area of high-performance scalable persistent object stores.

Master Yoda says I should be mindful of the future.
— ***Obi-Wan Kenobi***, *Jedi Apprentice*
But not at the expense of the present.
— ***Qui-Gon Jinn***, *Jedi Master*

Appendix A

The Sphere Public API

This appendix describes the Sphere public API. Section A.1 includes all the types used by the API, section A.2 defines the public basic calls, and section A.3 defines the public object operations. The API is included here to give the reader a flavour of how it is used. It is discussed in more detail in a technical report [Pri99b].

A.1 Types

This section describes all the types used by the Sphere public API.

A.1.1 `sphere_t`

It represents a Sphere store and is needed for every call in the API. No global data structures, associated with a particular store, are used by any of the API calls, so it is possible for a program to open two or more Sphere stores, associating each of them to a different variable of this type. Some limitations do exist though e.g. one program can only use different stores with the same implementation of regimes and kinds, since the regime and kind operations are global and cannot be changed for a specific store. The "constructor" for this type is the `sphere_new` call (↝§A.2.1) and instances of it are associated with a particular store with the `sphere_open` and `sphere_create` calls (↝§A.2.2 and §A.2.3 respectively).

A.1.2 `sphere_aux_data_t`

It represents application-specific data passed to the unswizzling callbacks (↝§A.1.18). The store does not know the structure of this data, only the user knows how to interpret it. It is defined as follows:

```
typedef void *sphere_aux_data_t;
```

A.1.3 `sphere_bool_t`

It represents a boolean value with values `SPHERE_FALSE` and `SPHERE_TRUE`.

A.1.4 sphere_desc_t

It represents a reference to the contents of a descriptor (↝§3.5.3). This is an abstract representation of descriptors, since the core of the store does not know their structure as they are defined by the implementation. It is defined as follows:

```
typedef unsigned char *sphere_desc_t;
```

Note: Each descriptor should be associated with a key. This key is used to efficiently retrieve the required descriptor inside a partition and should always appear as *the first field* of the descriptor (it is up to the application to ensure this). The type of this field is `sphere_desc_key_t` (↝§A.1.6).

A.1.5 sphere_desc_generator_t

It represents a callback which the store invokes when it needs to access the contents of a descriptor. Each of these callbacks is registered with the `sphere_setDescriptorGenerator` call (↝§A.2.11) and is associated with a particular object kind. The type is defined as follows:

```
typedef void
sphere_desc_generator_t (sphere_obj_info_t info,
                         sphere_desc_t *desc,
                         sphere_kind_t *kind,
                         sphere_size_t *size);
```

The `info` parameter is the one which is passed to the allocation call which causes the descriptor to be written in the partition. The generator returns `desc`, a reference to the descriptor contents, `kind`, which is the object kind of the descriptor, and `size`, which is the size of the descriptor contents in bytes.

Note: The descriptor generator callbacks are called fairly often (twice per descriptor allocated in each partition), therefore it is a good idea if they are implemented relatively efficiently, e.g. only returning a reference to the cached descriptor contents, rather than creating them every time.

A.1.6 sphere_desc_key_t

It represents the key with which descriptors can be looked up (↝§4.11.1).

A.1.7 sphere_dst_buffer_t

It represents a buffer where the recipient will write data to. It is defined as follows:

```
typedef unsigned char *sphere_dst_buffer_t;
```

A.1.8 sphere_err_t

It represents an error raised inside Sphere. Most API calls return a value of this type. A value of `sphere_ok` denotes that the call has been executed successfully. Any other value denotes that an error has occured. The `sphere_get_error_string` macro will return a string describing the error. Most of the time, the best course of action, when an error is raised, is to display the error string and halt the program. An example of how to handle errors is given below.

```
sphere_err_t err;
sphere_t     sphere;
sphere_pid_t pid;
...
err = sphere_setPersistentRoot(sphere, pid);
if (err != sphere_ok) {
  printf("Sphere Error %s\n", sphere_get_error_string(err));
  exit(-1);
}
```

A.1.9 sphere_hid_t

It represents a history, in terms of which updates take place atomically in the store (↝§4.4.1). This is created with the `sphere_createHistory` call (↝§A.2.8) and should be passed to any call which updates data in the store.

A.1.10 sphere_kind_t

It represents an object kind (↝§3.5.2 and §4.13.1).

A.1.11 sphere_obj_info_t

It represents the info field associated with an object which, in conjuction with the object kind, gives information on the type of the object (↝§4.5.4). Its interpretation is different for each object kind and depends on the kind's implementation.

A.1.12 sphere_offset_t

It represents an offset (typically into an object) and is unsigned.

A.1.13 sphere_pid_t

It represents a persistent identifier (PID) which uniquely identifies an object in the store (↝§3.7). A PID is created when an object is allocated, using the `allocate` operation (↝§A.3.2), and it must be passed to any operation involving that object. A PID can be re-used by the store *only* when the object that it references is de-allocated by the garbage collector. The value `SPHERE_NULL_PID` represents the null PID.

Note: Currently, PIDs are 32-bit wide.

A.1.14 sphere_regime_t

It represents a partition regime (↝§3.5.1 and §4.13.2).

A.1.15 sphere_size_t

It represents a size (typically in bytes) and is unsigned.

A.1.16 sphere_src_buffer_t

It represents a buffer where the recipient will read data from. It is defined as follows:

```
typedef unsigned char *sphere_src_buffer_t;
```

A.1.17 sphere_string_t

It represents a string. It is defined as follows:

```
typedef char *sphere_string_t;
```

A.1.18 sphere_unswizzling_callback_t

It represents a callback which the store invokes when it needs to unswizzle an object (↝§5.2.4). This unswizzling operation takes place directly on a buffer of the store disk cache for efficiency and the callback might be called several times for a single object, according to the way it spans over multiple buffers. Each of these callbacks is registered with the `sphere_setUnswizzlingCallback` call (↝§A.2.10) and is associated with a particular object kind. The type is defined as follows:

```
typedef void
sphere_unswizzling_callback_t (sphere_dst_buffer_t buffer,
                               sphere_offset_t offset,
                               sphere_size_t size,
                               sphere_aux_data_t aux);
```

The `buffer` parameter is a reference to the beginning of the subrange of the object which needs to be unswizzled (this subrange resides on a single buffer of the store disk cache), the `offset` and `size` parameters denote the beginning and size of the subrange in bytes (with an offset of 0 denoting the beginning of the object), and `aux` is application-specific data which the user can pass to the operation which will invoke this callback (`firstWrite`, `updateAll`, etc.). It must be emphasized that after the callback returns, the `buffer` parameter should not be accessed again, since the corresponding buffer might have been evicted from memory.

Note: These callbacks need to be reasonably efficient since they can be called multiple times per object. Also, note that the PID of the object is not passed to the callback. If the PID or an identity for the object needs to be used inside the callback, this should be included in the application-specific data.

A.1.19 sphere_word1_t

It represents an 1-byte unsigned word.

A.1.20 sphere_word2_t

It represents a 2-byte unsigned word.

A.1.21 sphere_word4_t

It represents a 4-byte unsigned word.

A.1.22 sphere_word8_t

It represents an 8-byte unsigned word.

A.2 API Calls

This section describes all the basic calls included in the Sphere public API. All object operations are described in section A.3. Notice that, if a value of 0 is passed to a parameter marked with †, Sphere will set this parameter to a default value.

A.2.1 sphere_new

Call	sphere_t sphere_new ()
MT-Safe	no
Parameters	
In	none
Out	none
Returns	reference to the newly-allocated store instance, NULL if the allocation fails

It allocates a new store instance and returns a reference to it. It also performs some initialisation on it, therefore it is important to *always* create a store instance using this call. A call to sphere_new must be immediately followed by a call to either sphere_create (↝§A.2.2) or sphere_open (↝§A.2.3) in order to associate the newly-allocated store instance with a physical store on disk.

A.2.2 sphere_create

Call	sphere_err_t sphere_create (sphere, configName, appName, appVersion, diskCacheSize)	
MT-Safe	It should be run in single-threaded mode, with no other calls taking place during it.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_string_t configName	name of the configuration file, including full path
	sphere_string_t appName	name of the application which is creating the store
	sphere_string_t appVersion	version of the application which is creating the store
	sphere_size_t diskCacheSize†	size of the disk cache in bytes
Out	none	
Returns	potential error	

It creates for the first time the store described by a configuration file with name configName. None of the segments of that store must exist (unless they are raw partitions — this also applies to the log segments), otherwise this operation will fail. The appName and appVersion parameters are used for fingerprinting the

store for future consistency checks. The `diskCacheSize` parameter denotes the size of the buffer which will be used for caching the pages of the store.

A.2.3 `sphere_open`

Call	`sphere_err_t` `sphere_open (sphere, configName, diskCacheSize, readOnly)`	
MT-Safe	It should be run in single-threaded mode, with no other calls taking place during it.	
Parameters		
In	`sphere_t sphere`	the store instance
	`sphere_string_t configName`	name of the configuration file, including full path
	`sphere_size_t diskCacheSize`†	size of the disk cache in bytes
	`sphere_bool_t readOnly`	read-only flag
Out	none	
Returns	potential error	

It opens an already-existing store, described by a configuration file with name `configName`. All the segments of that store must exist (this also applies to the log segments), otherwise this operation will fail. The `diskCacheSize` parameter denotes the size of the buffer which will be used for caching the pages of the store and the `readOnly` flag determines whether the store will be opened in read-only mode or not.

A.2.4 `sphere_close`

Call	`sphere_err_t` `sphere_close (sphere)`	
MT-Safe	It should be run in single-threaded mode, with no other calls taking place during it.	
Parameters		
In	`sphere_t sphere`	the store instance
Out	none	
Returns	potential error	

It closes the store by writing all the dirty pages and flushing the log. This call *must* be invoked before the application exits, in order to ensure that all updates have been propagated safely onto disk. After calling `sphere_close`, the `sphere` parameter should *not* be used again.

A.2.5 sphere_setPersistentRoot

Call	sphere_err_t sphere_setPersistentRoot (sphere, pid, hid)	
MT-Safe	no	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the new persistent root
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This call sets the persistent root of the store (↝§3.4.1 and §3.7.2). Sphere supports only *one* persistent root per store. Any hierarchical root naming must be done by the user.

Note: If this call is invoked during a stabilisation operation which allocates objects in the store, it must be called *after* all `firstWrite` operations (↝§A.3.6) have taken place.

A.2.6 sphere_getPersistentRoot

Call	sphere_err_t sphere_getPersistentRoot (sphere, pid)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
Out	sphere_pid_t *pid	the PID of the persistent root
Returns	potential error	

It retrieves the persistent root of the store (↝§3.4.1 and §3.7.2).

A.2.7 sphere_inspectObject

Call	sphere_err_t sphere_inspectObject (sphere, pid, kind, regime, size)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
Out	sphere_kind_t *kind	the kind of the object
	sphere_regime_t *regime	the regime on which the object resides
	sphere_size_t *size	the size of the object in bytes
Returns	potential error	

It retrieves enough information about the object with the given PID, in order to be able to call object operations on it (↝§A.3.1) and/or reserve enough memory in the in order to fetch it. The `kind` and `regime`

parameters are the kind of the object and the regime on which it currently resides and `size` is the size of the object in bytes.

A.2.8 sphere_createHistory

Call	`sphere_err_t` `sphere_createHistory (sphere, hid)`	
MT-Safe	yes	
Parameters		
In	`sphere_t sphere`	the store instance
Out	`sphere_hid_t *hid`	the history
Returns	potential error	

It creates a new history (⇝§4.4.1) in terms of which some updates will take place atomically in the store.

A.2.9 sphere_commitHistory

Call	`sphere_err_t` `sphere_commitHistory (sphere, hid)`	
MT-Safe	yes	
Parameters		
In	`sphere_t sphere`	the store instance
	`sphere_hid_t hid`	the history
Out	none	
Returns	potential error	

This call commits the updates which have taken place in terms of the `hid` parameter and flushes the log (⇝§4.4.1). If it completes successfully, then the updates are guaranteed to have been propagated onto the store.

Note: After invoking this operation, the `hid` parameter should not be used again, without being re-initialised with the `sphere_createHistory` call.

A.2.10 sphere_setUnswizzlingCallback

Call	`sphere_err_t` `sphere_setUnswizzlingCallback (sphere, kind, callback)`	
MT-Safe	no	
Parameters		
In	`sphere_t sphere`	the store instance
	`sphere_kind_t kind`	the object kind which the callback will be associated with
	`sphere_unswizzling_callback_t callback`	the unswizzling callback
Out	none	
Returns	potential error	

This renders `callback` to be the unswizzling callback (↝§5.2.4) associated with the `kind` parameter. Only one callback can be associated with an object kind. If no callback is set for a given kind, then none will be called when updates take place on objects of that kind.

Note: This call should only be invoked *after* a `sphere_create` or a `sphere_open` call and *before* any access to the store takes place.

A.2.11 `sphere_setDescriptorGenerator`

Call	`sphere_err_t` `sphere_setDescriptorGenerator (sphere,kind, generator)`	
MT-Safe	no	
Parameters		
In	`sphere_t sphere`	the store instance
	`sphere_kind_t kind`	the object kind which the generator will be associated with
	`sphere_desc_generator_t generator`	the descriptor generator
Out	none	
Returns	potential error	

This renders `generator` to be the descriptor generator associated with the `kind` parameter. Only one generator can be associated with an object kind. Generators should be set for *all* kinds which require descriptors.

Note: This call should only be invoked *after* a `sphere_create` or a `sphere_open` call and *before* any access to the store takes place.

A.3 Object Operations

This section describes the public object operations. They are invoked differently from the basic API calls, described in section A.2. Section A.3.1 describes the dispatching mechanism used.

A.3.1 Dispatching Mechanism

In order to invoke an operation on an object, the user, apart from the parameters to the operation, needs to know the object's kind and the regime where it resides. Given the first two and the name of the operation, the `SPHERE_KIND_OP` macro returns the appropriate function, to which the rest of the parameters are then passed. If either the kind or regime parameters are not known, they can be retrieved with the `sphere_inspectObject` call (↝§A.2.7). An example, which fetches an object with the `fetch` operation, is given below

```
sphere_err_t        err;
sphere_t            sphere;
sphere_pid_t        pid;
sphere_kind_t       kind;
sphere_regime_t     regime;
```

```
sphere_obj_info_t   info;
sphere_dst_buffer_t buffer;
...
err = sphere_inspectObject(sphere, pid, &kind, &regime, &size);
/* check error */
buffer = allocateSomeMemory(size);
err = SPHERE_KIND_OP(kind, regime, fetch)(sphere, pid, &info, buffer);
/* check error */
```

A.3.2 allocate

Call	sphere_err_t allocate (sphere, size, info, hid, pid)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
	sphere_size_t size	the size of the new object
	sphere_obj_info_t info	the info field of the new object
	sphere_hid_t hid	the history in terms of which this allocation will take place
Out	sphere_pid_t *pid	the PID of the newly-allocated object
Returns	potential error	

This operation allocates a new object in the store. It accepts the object's size and info field and returns the new PID for that object. Notice that the allocation does *not* write the contents of the object in the store. This is done later with the `firstWrite` operation (⇝§A.3.6).

Note: Operations on the newly allocated object can only take place *after* the history, in terms of which it was allocated, has been committed.

A.3.3 fetch

Call	sphere_err_t fetch (sphere, pid, info, buffer)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object to be fetched
Out	sphere_obj_info_t *info	the info field of the object
	sphere_dst_buffer_t buffer	the buffer where the object contents will be copied
Returns	potential error	

It fetches an object in its entirity from the store into memory and retrieves its info field. The `buffer` parameter should point to a location with enough free memory to hold the object.

A.3.4 partialFetch

Call	sphere_err_t partialFetch (sphere, pid, offset, size, buffer)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object to be fetched
	sphere_offset_t offset	the starting offset of the object's subrange
	sphere_size_t size	the size of the object's subrange
Out	sphere_dst_buffer_t buffer	the buffer where the object contents will be copied
Returns	potential error	

It fetches a subrange of an object from the store into memory. This subrange is specified by the `offset` and `size` parameters. The `buffer` parameter should point to a location with enough free memory to hold the specified subrange of the object.

A.3.5 getInfo

Call	sphere_err_t getInfo (sphere, pid, info)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object whose info field will be retrieved
Out	sphere_obj_info_t *info	the info field of the object
Returns	potential error	

It retrieves the info field of an object (⇝§4.5.4).

A.3.6 firstWrite

Call	sphere_err_t firstWrite (sphere, pid, size, info, buffer, aux, hid)	
MT-Safe	yes	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_size_t size	the size of the object
	sphere_obj_info_t info	the info field of the object
	sphere_src_buffer_t buffer	the buffer where the new contents of the object will be copied from
	sphere_aux_data_t aux	the application-specific data for the unswizzling callback
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation installs the contents of a newly-allocated object for the first time. It must be called after a call to the `allocate` operation (↝§A.3.2) and no other operations should be invoked on that object in between. Also, Sphere requires that all the `allocate` operations for a given history are grouped and are executed before all the `firstWrite` operations. Furthermore, all the `firstWrite` opreations *must* be invoked in the same order as the `allocate` operations. It is up to the user to ensure this. The `size`, `info`, and `hid` parameters must be the same as the ones used for the `allocate` operation which allocated the object.

The `buffer` parameter points to a memory location which contains the new contents of the object. If an unswizzling callback has been set for the kind of the object (↝§A.1.18 and §A.2.10), then it will be called, with the `aux` parameter being passed to it, after the new contents of the object have been copied to the buffer of the disk cache. This callback might be called multiple times, if the object spans over several pages. If the value of the `buffer` parameter is `NULL`, then it will be solely up to the unswizzling callback to fill in the contents of the object.

Note: The `firstWrite` operation must *only* be called once per object and `only` to install its contents for the first time. Operations on the newly allocated object can only take place *after* the history, in terms of which it was allocated, has been committed.

A.3.7 updateAll

Call	sphere_err_t updateAll (sphere, pid, buffer, aux, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_src_buffer_t buffer	the buffer where the new contents of the object will be copied from
	sphere_aux_data_t aux	the application-specific data for the unswizzling callback
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates an entire object. The buffer parameter points to a memory location which contains the new contents of the object. If an unswizzling callback has been set for the kind of the object (↝§A.1.18 and §A.2.10), then it will be called, with the aux parameter being passed to it, after the new contents of the object have been copied to the buffer of the disk cache. This callback might be called multiple times, if the object spans over several pages. If the value of the buffer parameter is NULL, then it will be solely up to the unswizzling callback to fill in the contents of the object.

A.3.8 updateRange

Call	sphere_err_t updateRange (sphere, pid, offset, size, buffer, aux, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_offset_t offset	the starting offset of the object's subrange
	sphere_size_t size	the size of the object's subrange
	sphere_src_buffer_t buffer	the buffer where the new contents of the object will be copied from
	sphere_aux_data_t aux	the application-specific data for the unswizzling callback
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates a subrange of an object. The offset and size parameters define the subrange and the buffer parameter points to a memory location which contains the new contents of the subrange. If an

unswizzling callback has been set for the kind of the object (⇝§A.1.18 and §A.2.10), then it will be called, with the aux parameter being passed to it, after the new contents of the object have been copied to the buffer of the disk cache. This callback might be called multiple times, if the object spans over several pages. If the value of the buffer parameter is NULL, then it will be solely up to the unswizzling callback to fill in the contents of the object.

A.3.9 updateRef

Call	sphere_err_t updateRef (sphere, pid, offset, value, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_offset_t offset	the offset of the field to be updated
	sphere_pid_t value	the new value for the field
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates a single reference field on an object. The offset parameter determines the location of this field and value is its new value.

Note: The offset parameter is required to be aligned according to the size of the pjsl_pid_t type.

A.3.10 update1

Call	sphere_err_t update1 (sphere, pid, offset, value, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_offset_t offset	the offset of the field to be updated
	sphere_word1_t value	the new value for the field
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates a single 1-byte field on an object. The offset parameter determines the location of this field and value is its new value.

A.3.11 update2

Call	sphere_err_t update2 (sphere, pid, offset, value, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_offset_t offset	the offset of the field to be updated
	sphere_word2_t value	the new value for the field
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates a single 2-byte field on an object. The `offset` parameter determines the location of this field and `value` is its new value.

Note: The `offset` parameter is required to be 2-byte aligned.

A.3.12 update4

Call	sphere_err_t update4 (sphere, pid, offset, value, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_offset_t offset	the offset of the field to be updated
	sphere_word4_t value	the new value for the field
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates a single 4-byte field on an object. The `offset` parameter determines the location of this field and `value` is its new value.

Note: The `offset` parameter is required to be 4-byte aligned.

A.3.13 update8

Call	sphere_err_t update8 (sphere, pid, offset, value, hid)	
MT-Safe	Concurrent updates on the same object are not safe and it is up to the user to serialize them. Concurrent updates on different objects are safe.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_pid_t pid	the PID of the object
	sphere_offset_t offset	the offset of the field to be updated
	sphere_word8_t value	the new value for the field
	sphere_hid_t hid	the history in terms of which this update will take place
Out	none	
Returns	potential error	

This operation updates a single 8-byte field on an object. The `offset` parameter determines the location of this field and `value` is its new value.

Note: The `offset` parameter is required to be 8-byte aligned.

I need to speak to the Jedi Council. The situation has become much more complicated.
— **Qui-Gon Jinn**, *Jedi Master*

Appendix B

The Sphere Engineering API

This appendix describes the Sphere engineering API. This is a small extension to the Sphere public API, described in appendix A, and incorporates facilities that are needed typically by browser-type programs. Section B.1 includes all the types used by the Sphere engineering API and section B.2 defines the engineering calls. The Sphere engineering API is discussed in more detail in a technical report [Pri99b].

B.1 Types

Apart from the types presented in section A.1, the Sphere engineering API also needs some additional ones. These are covered below.

B.1.1 `sphere_eng_obj_callback_t`

It represents a callback that is invoked once per object found by the `sphere_engFindObjects` call (⇝§B.2.3). The type is defined as follows:

```
typedef sphere_err_t
sphere_eng_obj_callback_t (sphere_pid_t pid,
                           sphere_kind_t kind,
                           sphere_obj_info_t info,
                           sphere_aux_data_t aux);
```

The `pid` parameter is the PID of the object, the `kind` parameter its kind, and the `info` parameter its info field (as returned by the `getInfo` operation, ⇝§A.3.5). Finally, the `aux` parameter is the application-specific data, passed to `sphere_engFindObjects`.

The callback should either return `sphere_ok` to indicate success or `sphere_err_iter_aborted` to indicate failure. In the latter case, the iteration will stop and the error will be returned by `sphere_engFindObjects` to its caller.

B.1.2 sphere_eng_part_callback_t

It represents a callback that is invoked once per partition found by the sphere_engFindPartitions call (↝§B.2.1). The type is defined as follows:

```
typedef sphere_err_t
sphere_eng_part_callback_t (sphere_partition_t part,
                            sphere_regime_t regime,
                            sphere_aux_data_t aux);
```

The part parameter is the identifier of the partition and the regime parameter is its regime. Finally, the aux parameter is the application-specific data, passed to sphere_engFindPartitions.

The callback should either return sphere_ok to indicate success or sphere_err_iter_aborted to indicate failure. In the latter case, the iteration over partitions will stop and the error will be returned by sphere_engFindPartitions to its caller).

B.1.3 sphere_partition_t

It represents a partition identifier that uniquely identifies a partition in the store (↝§3.4.3 and §3.6.1). This is not used by the standard Sphere public API, since the existence of partitions is hidden from it.

B.2 API Calls

This section describes all the calls of the Sphere engineering API.

B.2.1 sphere_engFindPartitions

Call	sphere_err_t sphere_engFindPartitions (sphere, regimes, length, callback, aux)	
MT-Safe	No.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_regime_t *regimes	the regime vector
	sphere_length_t length	the length of the regime vector
	sphere_eng_part_callback_t callback	the callback
	sphere_aux_data_t aux	the application-specific data for the callback
Out	none	
Returns	potential error	

It invokes the callback associated with the callback parameter, once per allocated partition in the store. If the length parameter is 0, the callback will be invoked for all partitions. If the length parameter is not 0, then it will be assumed that it is the length of a vector, pointed to by the regimes parameter. In the latter case, the callback will be invoked only for partitions that implement the regimes that appear in this vector.

Every time the callback is invoked, the aux parameter will be passed to it.

If the callback returns the value sphere_err_iter_aborted, the call will cease its operation and return the same error to its caller.

B.2.2 sphere_engFindDescriptor

Call	sphere_err_t sphere_engFindDescriptor (sphere, part, descKey, descPid)	
MT-Safe	No.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_partition_t part	the partition identifier
	sphere_desc_key_t descKey	the descriptor key
Out	sphere_pid_t *descPid	the PID of the descriptor
Returns	potential error	

It determines whether a partition contains a descriptor with a given key. The part parameter is the partition identifier and the descKey parameter is the descriptor key. The call returns the PID of the requested descriptor through the descPid parameter. If the descriptor is not found, then descPid is set to SPHERE_NULL_PID.

Note: This call should only be invoked on partitions that implement regimes allowed to manage descriptors.

B.2.3 sphere_engFindObjects

Call	sphere_err_t sphere_engFindObjects (sphere, part, kinds, length, callback, aux)	
MT-Safe	No.	
Parameters		
In	sphere_t sphere	the store instance
	sphere_partition_t part	the partition identifier
	sphere_kind_t *kinds	the kind vector
	sphere_length_t length	the length of the kind vector
	sphere_eng_obj_callback_t callback	the callback
	sphere_aux_data_t aux	the application-specific data for the callback
Out	none	
Returns	potential error	

It invokes the callback associated with the callback parameter, once per object in a partition. The identifier of the required partition is specified with the part parameter. If the length parameter is 0, the callback will be invoked for all objects. If the length parameter is not 0, then it will be assumed that it is the length of a vector, pointed to by the kinds parameter. In the latter case, the callback will be invoked only for objects with kinds that appear in this vector. Every time the callback is invoked, the aux parameter will be passed

to it.

If the callback returns the value `sphere_err_iter_aborted`, the call will cease its operation and return the same error to its caller.

Note: It is possible that some of the objects, retrieved with this call, are garbage and not transitively reachable from the persistent root of the store.

You can't stop change any more than you can stop the suns from setting.
— ***Shmi Skywalker***, *Slave*

Appendix C

The Sphere Configuration File

This appendix outlines the structure of the Sphere configuration file (↝§4.6.1), which mainly allows the user to specify where the log and store segments reside, as well as to set some other parameters. Section C.1 gives an overview of the parameters that can be set using the Sphere configuration file. Sections C.2, C.3, and C.4 include the sample configuration files distributed with the Sphere release, which are targetted for small, medium, and large memory footprints respectively. Finally, for illustration purposes, section C.5 includes the configuration file used for all the promotion experiments (↝§5.3).

C.1 Overview of Parameters

The Sphere configuration file is comprised of four sections: *Session*, *Log*, *Recovery*, and *Store*, each of which contains a number of parameters. The section headings should appear in the configuration file in the form

```
<section_heading>
```

and the parameters in the form

```
parameter_name = value(s)
```

The sections are described below, along with the parameters they contain.

- **Session** — It contains any parameters related to the current invocation of Sphere.
  - `readOnly = ( <false> | <true> )`
    It specifies whether or not the store will be opened in read-only mode. If it is opened in read-only mode, then the log and store segments are opened in read-only mode (hence, they can be accessed from read-only devices, e.g. CD-ROM, if necessary) and no updates to the store are allowed.
- **Log** — It contains any parameters related to the log manager (↝§4.4.2).

- `logName = ( <full path to file> | <raw partition name> )`
  It specifies the location of the log segment.
  Example: `logName = /home/darth_vader/stores/deathstar_blueprints.pjl`
- `logSize = <size>`
  It specifies the size of the log segment.
- `cacheSize = <size>`
  It specifies the size of the caches used by the log manager cursors [Ham00].
- `cacheNumber = <number>`
  It specifies the number of the caches used by the log manager cursors [Ham00].

❑ **Recovery** — It contains any parameters related to the recovery manager (⇝§4.4.2).

- `debugRecords = ( <false> | <true> )`
  It specifies whether some extra log records, included for debugging purposes, should be generated by the recovery manager.

❑ **Store** — It contains any generic store parameters.

- `segment = <id> ( <full path to file> | <raw partition name> ) <max size>`
  It specifies the SID, location, and maximum size of a store segment (⇝§3.4.4). If a raw partition is used, the maximum size parameter is ignored and the size of the raw partition is used instead. At least one `segment` parameter is required per store segment.
  Example: `segment = 0 /home/darth_vader/stores/deathstar_blueprints.pjs 32m`
- `cacheSize = <size>`
  It specifies the size of the disk cache (⇝§4.7).

More parameters might be added to the Sphere configuration file, if required in the future.

C.2 "Small" Sphere Sample Configuration File

```
##########################################
##   Sample SMALL Config file for Sphere
##
##   Tony Printezis, Oct 1999
##########################################

## There are four possible size formats:
##
##   bytes  1073741824
##   KBs       1048576k
##   MBs          1024m
##   GBs             1g
##

## You are recommended, if you can, to attempt to place each log/store
## segment on different physical disks for increased performance.
## If you can afford it, try to place at least the log segment on a disk
## on its own, so that head movement caused by other disk traffic does
## not interfere with it.

<Session>

## Determines whether the store should be opened in read-only mode or not
## Possible values: true / false

  readOnly      = false


<Log>

## Full path to the log segment - this can be a raw partition
## Suggested extension: .pjl

  logName       = /home/darth_vader/stores/deathstar_blueprints.pjl


## The size of the log segment
##
## You'll have to increase this, if you do large stabilisations

   logSize      = 10m


## Size of the log tail
##
```

```
## Increasing this can improve the performance of large updates.

  cacheSize      = 256k


## Number of log buffers
##
## This is used to open different cursors during mainly recovery.
## A small-ish number should be sufficient for most cases.

  cacheNumber    = 4


<Recovery>

## Generate complimentary log traffic (e.g. timestamps)
## (for improved performance ensure this feature is set to false)

  debugRecords = false


<Store>

## There are the store segments - they should be in the form:
##
##     segment = <number> <file/raw partition name> <max size>
##
## <number> is the segment id and must be unique per segment.
##    There must be at least one segment with id 0 (boot segment).
##    The maximum of this is 31 (maximum 32 segments)
##
## <file/raw partition name> full path to the file/raw partition
##    Suggested extension: .pjs
##
## <max size> maximum size that the segment can grow to. This must be
##    at least 2m and up to 1g.
##    If the segment is a raw partition, <max size> is ignored and Sphere
##    will automatically use up the entire raw partition
##
##    NOTE: This size can be increased, to deal with a segment running
##    out of space.
##
## Addition of segments is also supported, to deal with a store running
## out of space.
##
## Currently, multiple segments are supported in order to facilitate the
## use of stores over 2GB and multiple disks. Try to keep the number of
```

```
## segments to a minimum.


## Segment over a file

    segment      = 0 /home/darth_vader/stores/deathstar_blueprints.pjs  32m

## Segment over a raw partition
##
## Uncomment this if you want to use it - you can also use it as segment
## 0 if necessary

##  segment      = 1 /dev/rdsk/c0t0d0s4                                  0


## The size of the disk cache, i.e. where pages from the store are cached
##
## Size matters. The bigger the better (obviously, there is not point
## in running with a disk cache larger than the store itself).
## Also, the disk cache should be large-ish, if raw partitions are used
## for store segments (since they are not cached by the OS).

    cacheSize    = 2m
```

C.3 "Medium" Sphere Sample Configuration File

```
##########################################
##   Sample MEDIUM Config file for Sphere
##
##   Tony Printezis, Oct 1999
##########################################

## There are four possible size formats:
##
##   bytes  1073741824
##   KBs       1048576k
##   MBs          1024m
##   GBs             1g
##

## You are recommended, if you can, to attempt to place each log/store
## segment on different physical disks for increased performance.
## If you can afford it, try to place at least the log segment on a disk
## on its own, so that head movement caused by other disk traffic does
## not interfere with it.

<Session>

## Determines whether the store should be opened in read-only mode or not
## Possible values: true / false

  readOnly      = false


<Log>

## Full path to the log segment - this can be a raw partition
## Suggested extension: .pjl

  logName       = /home/darth_vader/stores/deathstar_blueprints.pjl


## The size of the log segment
##
## You'll have to increase this, if you do large stabilisations

   logSize      = 100m


## Size of the log tail
##
```

```
## Increasing this can improve the performance of large updates.

  cacheSize      = 512k


## Number of log buffers
##
## This is used to open different cursors during mainly recovery.
## A small-ish number should be sufficient for most cases.

  cacheNumber    = 4


<Recovery>

## Generate complimentary log traffic (e.g. timestamps)
## (for improved performance ensure this feature is set to false)

  debugRecords = false


<Store>

## There are the store segments - they should be in the form:
##
##     segment = <number> <file/raw partition name> <max size>
##
## <number> is the segment id and must be unique per segment.
##    There must be at least one segment with id 0 (boot segment).
##    The maximum of this is 31 (maximum 32 segments)
##
## <file/raw partition name> full path to the file/raw partition
##    Suggested extension: .pjs
##
## <max size> maximum size that the segment can grow to. This must be
##    at least 2m and up to 1g.
##    If the segment is a raw partition, <max size> is ignored and Sphere
##    will automatically use up the entire raw partition
##
##    NOTE: This size can be increased, to deal with a segment running
##    out of space.
##
## Addition of segments is also supported, to deal with a store running
## out of space.
##
## Currently, multiple segments are supported in order to facilitate the
## use of stores over 2GB and multiple disks. Try to keep the number of
```

```
## segments to a minimum.

## Segment over a file

    segment      = 0 /home/darth_vader/stores/deathstar_blueprints.pjs 256m

## Segment over a raw partition
##
## Uncomment this if you want to use it - you can also use it as segment
## 0 if necessary

##  segment      = 1 /dev/rdsk/c0t0d0s4                                    0


## The size of the disk cache, i.e. where pages from the store are cached
##
## Size matters. The bigger the better (obviously, there is not point
## in running with a disk cache larger than the store itself).
## Also, the disk cache should be large-ish, if raw partitions are used
## for store segments (since they are not cached by the OS).

    cacheSize    = 10m
```

C.4 "Large" Sphere Sample Configuration File

```
##########################################
##   Sample LARGE Config file for Sphere
##
##   Tony Printezis, Oct 1999
##########################################

## There are four possible size formats:
##
##   bytes  1073741824
##   KBs       1048576k
##   MBs          1024m
##   GBs             1g
##

## You are recommended, if you can, to attempt to place each log/store
## segment on different physical disks for increased performance.
## If you can afford it, try to place at least the log segment on a disk
## on its own, so that head movement caused by other disk traffic does
## not interfere with it.

<Session>

## Determines whether the store should be opened in read-only mode or not
## Possible values: true / false

  readOnly      = false


<Log>

## Full path to the log segment - this can be a raw partition
## Suggested extension: .pjl

  logName       = /home/darth_vader/stores/deathstar_blueprints.pjl


## The size of the log segment
##
## You'll have to increase this, if you do large stabilisations

   logSize      = 300m


## Size of the log tail
##
```

```
## Increasing this can improve the performance of large updates.

  cacheSize      = 1m


## Number of log buffers
##
## This is used to open different cursors during mainly recovery.
## A small-ish number should be sufficient for most cases.

  cacheNumber    = 4


<Recovery>

## Generate complimentary log traffic (e.g. timestamps)
## (for improved performance ensure this feature is set to false)

  debugRecords = false


<Store>

## There are the store segments - they should be in the form:
##
##     segment = <number> <file/raw partition name> <max size>
##
## <number> is the segment id and must be unique per segment.
##    There must be at least one segment with id 0 (boot segment).
##    The maximum of this is 31 (maximum 32 segments)
##
## <file/raw partition name> full path to the file/raw partition
##    Suggested extension: .pjs
##
## <max size> maximum size that the segment can grow to. This must be
##    at least 2m and up to 1g.
##    If the segment is a raw partition, <max size> is ignored and Sphere
##    will automatically use up the entire raw partition
##
##    NOTE: This size can be increased, to deal with a segment running
##    out of space.
##
## Addition of segments is also supported, to deal with a store running
## out of space.
##
## Currently, multiple segments are supported in order to facilitate the
## use of stores over 2GB and multiple disks. Try to keep the number of
```

```
## segments to a minimum.

## Segment over a file

    segment      = 0 /home/darth_vader/stores/deathstar_blueprints.pjs  1g

## Segment over a raw partition
##
## Uncomment this if you want to use it - you can also use it as segment
## 0 if necessary

## segment      = 1 /dev/rdsk/c0t0d0s4                                  0


## The size of the disk cache, i.e. where pages from the store are cached
##
## Size matters. The bigger the better (obviously, there is not point
## in running with a disk cache larger than the store itself).
## Also, the disk cache should be large-ish, if raw partitions are used
## for store segments (since they are not cached by the OS).

    cacheSize    = 20m
```

C.5 Sphere Configuration File for Promotion Experiments

```
########################################
## Config File for Promotion Experiments
########################################

<Session>

  readOnly       = false


<Log>

  logName        = /dev/rdsk/c0t1d0s3
  logSize        = 1900m
  cacheSize      = 512K
  cacheNumber    = 4


<Recovery>

  debugRecords   = false


<Store>

  segment        = 0 /dev/rdsk/c0t0d0s4 0
  cacheSize      = 0
  # the benchmarks will specify this
```

Hokey religions and ancient weapons are no match for a good blaster at your side, kid.
— ***Han Solo**, Correlian Smuggler*

Appendix D

SGGC User's Guide

This appendix includes the user's guide of the off-line Sphere garbage collector and its two associated utilities. Section D.1 describes SGM, the Sphere Global Marker, section D.2 describes SGS, the Sphere Global Sweeper, and section D.3 describes SGGC, the Sphere Global Garbage Collector itself. Finally, section D.4 includes sample output from an invocation of SGGC, for illustration purposes.

D.1 SGM — Sphere Global Marker

As described in section 6.6, SGM performs a global marking phase on a Sphere store and marks as garbage (i.e. nuked) all objects unreachable from the persistent root, so that they can be reclaimed by the concurrent Sphere GC (or by SGS, ⇝§D.2). These include all cross-partition garbage cycles. The operation of SGM is fault-tolerant and the nuking of all objects atomic, i.e. if there is a crash during the operation of SGM the state of the store will be reverted back to what it was just before SGM was launched.

Launching SGM with no parameter yields its usage.

```
[sparc]{tony@neptune}302: sgm
== Sphere Global Marker (SGM), October 1999

  usage: sgm -config <name>  [ optional ]
    -config <name>
            full path to the Sphere config file

    [ optional ]

    -diskcache <size>
            disk cache size in MBs
           (default: 10 MBs)
    -verbose   <level>
            level of info displayed
```

```
          0=quiet        1=verbose      2=timer        3=stats
          4=mark trace   5=nuke trace   6=all traces
        (default: 1)
  -readonly
         if set, prints stats only
```

Its parameters are described below.

- `-config <name>` (compulsory) — `<name>` is the full path to the Sphere configuration file.
- `-diskcache <size>` (optional) — `<size>` is the size of the disk cache in MBs. The default value is 10MB.
- `-verbose <level>` (optional) — `<level>` determines the level of verboseness. Its possible values are: 0 - quiet, 1 - some messages, 2 - timing information, 3 - statistics, 4 - marking trace, 5 - nuking trace, 6 - all traces. The default value is 1.
- `-readonly` (optional) — if it is set, then SGM does not nuke any objects but instead only generates statistics. The default is off.

Example:

```
sgm -config /home/tony/store.cfg -diskcache 5 -verbose 3 -readonly
```

D.2 SGS — Sphere Global Sweeper

As described in section 6.6, SGS calls the partition GC on every partition in a Sphere store. It assumes that SGM has been invoked first and has nuked all garbage objects in the store. Its operation is fault-tolerant, i.e. the store is not left in an inconsistent state if there is a failure during its operation. However, the atomicity of SGS is at the partition level, i.e. each partition will be sweeped atomically but *not* the entire store. For example, after an SGS crash, objects reclaimed in partitions that had been operated on before the crash, will *not* be re-instated.

Launching SGS with no parameter yields its usage.

```
[sparc]{tony@neptune}303: sgs
== Sphere Global Sweeper (SGS), October 1999

  usage: sgs -config <name>  [ optional ]
    -config <name>
           full path to the Sphere config file

    [ optional ]

    -diskcache <size>
           disk cache size in MBs
          (default: 10 MBs)
    -verbose   <level>
```

```
        level of info displayed
        0=quiet   1=verbose   2=timer
       (default: 1)
```

Its parameters are described below.

- `-config <name>` (compulsory) — `<name>` is the full path to the Sphere configuration file.
- `-diskcache <size>` (optional) — `<size>` is the size of the disk cache in MBs. The default value is 10MB.
- `-verbose <level>` (optional) — `<level>` determines the level of verboseness. Its possible values are: 0 - quiet, 1 - some messages, 2 - timing information. The default value is 1.

Example:

```
sgs -config /home/tony/store.cfg -diskcache 5 -verbose 3
```

D.3 SGGC — Sphere Global Garbage Collector

As described in section 6.6, SGGC performs an off-line global garbage collection of a Sphere store. Information on the fault-tolerance of each phase of SGGC is given in sections D.1 and D.2.

Launching SGGC with no parameter yields its usage.

```
[sparc]{tony@neptune}304: sggc
== Sphere Global Garbage Collector (SGGC), October 1999

  usage: sggc -config <name>  [ optional ]
    -config <name>
           full path to the Sphere config file

    [ optional ]

    -diskcache <size>
           disk cache size in MBs
          (default: 10 MBs)
    -verbose   <level>
           level of info displayed
             0=quiet        1=verbose      2=timer        3=stats
             4=mark trace   5=nuke trace   6=all traces
          (default: 1)
```

Its parameters are described below.

- `-config <name>` (compulsory) — `<name>` is the full path to the Sphere configuration file.
- `-diskcache <size>` (optional) — `<size>` is the size of the disk cache in MBs. The default value is 10MB.

- `-verbose <level>` (optional) — `<level>` determines the level of verboseness. Its possible values are: 0 - quiet, 1 - some messages, 2 - timing information, 3 - statistics, 4 - marking trace, 5 - nuking trace, 6 - all traces. The default value is 1.

Example:

```
sggc -config /home/tony/store.cfg -diskcache 5 -verbose 3
```

D.4 SGGC — Sample Output

For illustration purposes, the output from a run of SGGC has been included below.

```
[sparc]{tony@neptune}305: sggc -config /home/tony/store.cfg -diskcache 5 -verbose 3
== Sphere Global Garbage Collector (SGGC), October 1999

== Marking Phase
== Nuking Phase

== TIMING STATS

--   Marking Phase             11.501621 secs
--   Nuking Phase              8.45785 secs
--   Total                     19.547432 secs

== SPHERE GLOBAL MARKER STATS

--   Live Partitions              14  (100.00%)
--   Garbage Partitions            0  (  0.00%)
--   Total Partitions             14

--   Live Objects             160001  ( 66.67%)
--   Garbage Objects           80001  ( 33.33%)
--      * Nuked                80001  ( 33.33%)
--      * Dead                     0  (  0.00%)
--      * Other                    0  (  0.00%)
--   Total Objects            240002

--   Live Size               6.11 MB  ( 66.66%)    6253.12 KB     6403200 bytes
--   Garbage Size            3.05 MB  ( 33.34%)    3128.12 KB     3203200 bytes
--   Total Size              9.16 MB               9381.25 KB     9606400 bytes

== MARKING STACK STATS

--   Number of segments                1
--   Entries per segment           65536
--   Maximum entries on stack        400
```

```
== ALLOCATOR STATS

--   Partition Table            0.25 MB       256.00 KB       262144 bytes
--   Bitmaps                    0.05 MB        56.00 KB        57344 bytes
--   Marking Stack              0.25 MB       256.01 KB       262152 bytes
--   Misc                       0.00 MB         0.03 KB           28 bytes

--   Total                      0.55 MB       568.04 KB       581668 bytes

== Sweeping

-- Collecting Partition     0, Regime   0
--         GC Time             1.856489 secs
-- Collecting Partition     1, Regime   0
--         GC Time             1.708675 secs
-- Collecting Partition     2, Regime   0
--         GC Time             1.783396 secs
-- Collecting Partition     3, Regime   0
--         GC Time             1.584384 secs
-- Collecting Partition     4, Regime   0
--         GC Time             1.493131 secs
-- Collecting Partition     5, Regime   0
--         GC Time             1.493149 secs
-- Collecting Partition     6, Regime   0
--         GC Time             1.435067 secs
-- Collecting Partition     7, Regime   0
--         GC Time             1.484881 secs
-- Collecting Partition     8, Regime   0
--         GC Time             1.791699 secs
-- Collecting Partition     9, Regime   0
--         GC Time             0.962087 secs
-- Collecting Partition    10, Regime   0
--         GC Time             0.912439 secs
-- Collecting Partition    11, Regime   0
--         GC Time             0.912375 secs
-- Collecting Partition    12, Regime   0
--         GC Time             0.887552 secs
-- Collecting Partition    13, Regime   0
--         GC Time             0.66578 secs

-- Visited 14 Partitions
--      Total Time             18.492423 secs
```

Yo bana pee ho-tah, meendee ya (it translates to "your friend is a foolish one, methinks").
— **Watto**, *Toydarian Junk Dealer*

Appendix E

Glossary

A

Active History, *page 49*

A history that has not been committed and updates to the store in terms of it are still taking place.

Actively Allocating Partition Table (AAPT), *page 68*

A table that contains one entry per partition regime and denotes which partition will first attempt to satisfy an object-allocation request for that regime.

ARIES Algorithm, *page 51*

A recovery algorithm for storage systems (POSs, RDBs, etc.). It assumes a write-ahead logging protocol and guarantees consistency by reading the log and redoing committed updates that had not been propagated to disk and undoing uncommitted updates that had been propagated to disk.

B

Basic Block (BB), *page 43*

Each Sphere segment is split into a number of fixed-size blocks called Basic Blocks. When a partition needs to be allocated in a segment, a number of contiguous BBs is reserved for it. It follows that BBs are essentially the unit of allocation for partitions.

Basic Block Identifier (BBID), *page 58*

An identifier used to uniquely address BBs.

Bytecode, *page 9*

An intermediate representation of a program that can be executed efficiently by an interpreter. It is used to make executables architecture-independent, as only the interpreter needs to be ported.

C

Clock Algorithm, *page 60*

A page-replacement algorithm that approximates a least-recently-used policy but is typically cheaper to implement.

Compare-And-Swap (CAS) Instruction, *page 47*

A CPU instruction that atomically compares a value with the contents of a memory address and, if they are equal, overwrites the memory address with a second value. It is extensively used for locking purposes in multi-threaded environments. It is also referred to as the *test-and-set* instruction.

D

Descriptor, *page 34*

A special Sphere object that describes the contents of other objects (in particular, the location of reference fields). Descriptors are replicated in different partitions and shared by all objects of the same type in each partition.

Descriptor Location Cache (DLC), *page 79*

A main-memory cache, mapping descriptor keys to their PIDs.

Descriptor Table (DT), *page 78*

A hash table, rooted on the header of partitions that need descriptors, denoting which descriptors are allocated in that partition.

Disk Cache (DC), *page 59*

A pool of buffers that caches disk pages.

Disk Garbage Collection, *page 116*

The operation of identifying and reclaiming unreachable (garbage) objects in a POS.

E

Empty Object, *page 131*

Special objects that provide padding in partitions of small-object regimes.

Exact Memory Management, *page 21*

Memory management when the exact location of references on the runtime stacks is known. It is also referred to as *accurate*, *cooperative*, or *non-conservative*.

Exact VM (EVM) [WG99], *page 21*

A high-performance Java virtual machine from Sun Microsystems Inc. with a high-performance JIT compiler, fast thread synchronisation, and provision for exact memory management.

Explicit Stabilisation, *page 19*

A user-initiated stabilisation that propagates all updates on cached persistent objects to disk.

F

Forwarding Reference (FR), *page 40*

A scheme of moving an object from one partition to another and overwriting the offset field of its indirectory entry with its new PID so that any requests for the old PID will be forwarded to the new object location.

Fragmentation [JW98], *page 42*

The paradox when, even though enough memory or storage is available in total to satisfy an allocation request, there is not a single contiguous area large enough to satisfy the request.

G

Garbage Collection (GC) [Jon96, Wil92], *page 116*

The automatic identification and reclamation of unused (garbage, unreachable, etc.) memory or storage so that it can be recycled.

Ghosted Allocation, *page 88*

An object promotion algorithm designed to allocate large numbers of objects to a POS, efficiently and with minimal log traffic.

H

Handle, *page 19*

A small immutable data structure that provides one level of indirection to an object. It is used to allow objects to be moved for compaction reasons without their identity (i.e. the address of the handle) changing.

History, *page 49*

The Sphere-equivalent of a transaction.

History Table, *page 49*

An main-memory table containing information about each active history.

I

Implicit Stabilisation, *page 19*

A system-initiated stabilisation at the end of the execution of a PJama program that propagates all updates on cached persistent objects to disk.

Indirectory, *page 38*

A table that contains the indirectory entries of a partition.

Indirectory Entry, *page 38*

It provides one level of indirection to an object's contents, so that the object can be easily moved inside a partition without its PID (which contains the indirectory entry index) changing.

Indirectory Free-List, *page 54*

A list of freed indirectory entries.

Info Field, *page 55*

The info field is retrieved from an object along with its contents and contains kind-specific information, typically on the type of the object.

J

Java Programming Language [GJS96, AG96], *page 8*

A very popular object-oriented, machine-independent, garbage-collected, buzzword-compliant, programming language developed at Sun Microsystems Inc.

Java Platform [Sun98c], *page 9*

The Java platform refers to the Java programming language as well as all its standard libraries.

Java Development Kit (JDK), *page 9*

Sun's standard implementation of the Java platform which is distributed freely.

Java Language Specification (JLS) [GJS96, AG96], *page 9*

The formal specification of the Java programming language.

Java Object Serialisation (JOS) [Sun98b], *page 17*

A standard Java facility that can serialise object graphs into and deserialise them from bytestreams. Such bytestreams can be sent over the network (with the use of RMI), stored on disk, etc.

Just-In-Time Compiler (JIT), *page 21*

A compiler that compiles bytecodes into native code on-the-fly as they are executed by the virtual machine, in order to increase the system's performance.

K

Kind (Object Kind), *page 32*

An object kind represents a group of objects of the same type and behaviour.

L

Latch, *page 47*

A light-weight lock that allows threads to achieve exclusive access to data structures in an MT-safe way.

Log, *page 49*

A space on disk where updates to the store are "logged" for recovery purposes.

Log Manager, *page 49*

The module that manages the log in Sphere.

Log Record, *page 49*

A record written to the log that describes a store update. Upon recovery, log records are read in order to redo committed updates and undo uncommitted ones.

Log Sequence Number (LSN), *page 50*

An LSN is effectively an identifier that can uniquely address log records.

Logical Update, *page 52*

An update to the store that is *not* idempotent to being redone or undone.

M

Multi-Threaded Safety (MT-Safety), *page 47*

The guarantee that, if a data structure is accessed and updated by several threads, it will not be corrupted because of the multi-threaded accesses.

Mutator, *page 25*

The application that provides the load for Sphere.

Mutex, *page 47*

A lock that allows threads to exclusive exclusive access to data structures in an MT-safe way.

N

No-Steal Policy, *page 19*

Policy of a recovery system *not* to allow dirty pages to be written to the store without the corresponding transaction having been fully committed.

Nuking, *page 145*

Marking an object in the store as de-allocated. It is used by SGM to explicitly de-allocate cross-partition garbage cycles.

O

Oak [Gos93], *page 8*

The original name of the Java programming language.

Object, *page 28*

The unit of transfer between the mutator and Sphere.

Object, Large, *page 29*

Objects that are larger than a store page.

Object, Small, *page 29*

Objects that are smaller than a store page.

Object-Fault, *page 19*

The operation of moving objects from the store into the mutator heap.

Object Location Cache (OLC), *page 74*

A main-memory cache that maps PIDs to the store locations where the corresponding objects are stored. Unfortunately, it did not pay-off performance-wise and was eventually removed from Sphere.

Object-to-Indirectory Entry Table (OIET), *page 125*

A main-memory table, used by the Sphere compacting garbage collector, that maps object offsets to indirectory indexes.

Object-Oriented Database (OODB), *page 2*

A complete object database system, comprising query facilities, language mappings, etc.

Object Space, *page 53*

The space in a partition where the object contents are stored.

Open Persistent System, *page 16*

A persistent system that can handle external state through a well-defined interface.

Orthogonal Persistence (OP) [AM95], *page 10*

A language-independent model of persistence, defined by the principles of type orthogonality, persistence by reachability, and persistence independence.

P

Page, *page 29*

The unit of transfer between Sphere and the physical disks.

Page Descriptor (PD), *page 60*

A small data structure that describes a buffer of the Sphere disk cache.

Partition, *page 30*

An object container in Sphere that is also garbage collection and schema evolution unit.

Partition Identifier (PI), *page 37*

A unique identifier for partitions. The addressing scheme is logical and a PI does not depend on the partition location in the store.

Partition Location Cache (PLC), *page 72*

A main-memory cache that maps PIs to the store locations where the corresponding partitions are stored. It is also used for synchronisation between the mutator and the concurrent disk garbage collector.

Partition Table (PT), *page 37*

A persistent data structure that contains an entry per partition in the store and provides one level of indirection to its location. It is also used for locking partitions for updates, promotion, and garbage collection.

Partition Table Entry (PTE), *page 64*

An entry of the partition table.

Partition Table Directory (PTD), *page 67*

A persistent data structure, constantly pinned into memory, that provides more efficient access to the partition table.

Persistence, *page 9*

The concept of retaining data across multiple invocations of an application.

Persistence by Reachability, *page 10*

A concept of orthogonal persistence, according to which the lifetime of all objects is determined by reachability from a designated set of root objects, the Persistent Roots.

Persistence Independence, *page 11*

A concept of orthogonal persistence, according to which it is indistinguishable whether code is operating on short-lived or long-lived data.

Persistent EVM (PEVM), *page 21*

A port of the PJama system, based on the EVM interpreter. This is also referred to as PJama_1.

Persistent Identifier (PID), *page 29*

An identifier used to uniquely address objects in the store. A logical addressing scheme is employed, i.e. a PID does not depend on the physical location of the corresponding object inside the store.

Persistent Object Store (POS), *page 2*

The low-level technology of storage systems that use objects as their units of transfer between them and the application.

Persistent Root, *page 10*

An object in the store that is considered reachable, hence live, by default. All other objects transitively reachable from the persistent root are also considered live and will be retained by the Sphere garbage collector.

PJama [ADJ+96, AJDS96, Jor96a, JA98, AJ99b, Pri99a], *page 12*

An orthogonally persistent environment for the Java programming language.

PJama Classic, *page 18*

The prototype implementation of the PJama system, based on the Classic JDK interpreter.

Physical Update, *page 51*

An update to the store that is idempotent to being redone or undone.

Promotion (Object Promotion), *page 20*

The operation of allocating objects in the store and copying their contents from the mutator heap.

R

Recoverable Virtual Memory (RVM) [MS94], *page 19*

A package that provides generic logging and recovery facilities.

Recovery, *page 48*

The operation of ensuring that a store is in a consistent state, following a crash or failure, by undoing uncommitted updates and redoing committed updates.

Recovery Manager, *page 50*

The Sphere module that reads the log and decided whether recovery should take place, which updates should be redone, and which updates should be undone.

Reference Count (RC), *page 75*

A count associated with each object, denoting the number of cross-partition references that point to the object. It is used for the purposes of garbage collection.

Reference Count Update Buffer (RCUB), *page 76*

A main-memory buffer used to buffer reference count updates and applies them to the store in bulk in an attempt to optimise the store access pattern caused by such updates.

Regime (Partition Regime), *page 31*

A partition regime represents a group of partitions that contain similar objects, in size and behaviour, and are managed by the same policies (e.g. free-space management).

Relational Database (RDB), *page 2*

A database system that uses the relational model to store and query data.

Residency Check, *page 19*

Small pieces of code planted throughout a persistent system that check whether a dereferenced reference field corresponds to a fetched object and, if it doesn't, initiate an object-fault.

Resident Object Table (ROT), *page 19*

A main-memory table in the PJama systems that maps PIDs to memory addresses of cached objects.

S

Schema Evolution, *page 146*

The operation of evolving objects in a POS (i.e. their type, contents, and implementation) to reflect changes in the persistent application that uses it.

Segment, *page 30*

A partition container in Sphere that resides entirely on the same physical storage device.

Segment Table (ST), *page 56*

A main-memory table that contains information about all the segments used by the store.

Segment Identifier (SID), *page 56*

An identifier that uniquely addresses segments in a Sphere store.

Sphere [PAD+97b, PAD+97a, PAD98], *page 25*

The new persistent object store from the Department of Computing Science at the University of Glasgow, Scotland, described throughout this thesis.

Sphere Global Garbage Collector (SGGC), *page 144*

An off-line utility that performs a global garbage collection in order to reclaim all unreachable objects from a Sphere store.

Sphere Global Marker (SGM), *page 145*

An off-line utility that performs only the marking phase of SGGC.

Sphere Global Sweeper (SGS), *page 145*

An off-line utility that performs only the sweeping phase of SGGC.

Stabilisation, *page 19*

The operation of propagating updates from main-memory cached objects to the POS. In systems that implement persistence by reachability, this will also allocate to the POS newly-reachable objects.

Steal Policy, *page 51*

Policy of a recovery system to allow dirty pages to be written to the store before the corresponding transaction has been fully committed.

Swizzling, *page 19*

The operation of overwriting PIDs with the corresponding memory addresses in main-memory cached objects.

T

Transient Data, *page 16*

Data that only resides in main-memory and does not have a corresponding store image.

Type Orthogonality, *page 10*

A concept of orthogonal persistence, according to which persistence is available for all data, irrespective of type.

U

Ullage, *page 53*

The part of a partition that resides between the indirectory and the object space and provides space for expansion for these two spaces.

V

Virtual Log, *page 50*

A collection of log records stored in the log that correspond to a set of updates that have to be performed atomically.

Virtual Machine (VM), *page 13*

An application that executes a program that has been compiled to bytecode. Essentially, it provides a virtual architecture that matches the bytecode format.

W

Write-Ahead Logging (WAL), *page 49*

The policy of first forcing the corresponding log records to disk before some updates can be propagated to the store.

Who's the more foolish, the fool, or the fool who follows him?
*— **Obi-Wan Kenobi**, Jedi Master*

Bibliography

[ABC+83] 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, 1983.

[ABD+92] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In Bancilhon et al. [BDK92], pages 25–42.

[ACC82] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[ACC83a] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. Algorithms for a Persistent Heap. *Software — Practice and Experience*, 13(3):259–272, March 1983.

[ACC83b] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. CMS — A Chunk Management System. *Software — Practice and Experience*, 13(3):273–285, March 1983.

[AD97] O. Agesen and D. Detlefs. Finding References in Java™ Stacks. In *Proceedings of the OOPSLA'97 Workshop on Garbage Collection and Memory Management*, Atlanta, GA, USA, October 1997.

[ADG+99a] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of OOPSLA'99*, pages 207–222, Denver, Colorado, USA, November 1999.

[ADG+99b] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. Technical Report TR-99-76, Sun Microsystems Laboratories, Palo Alto, CA, April 1999.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(4):68–75, December 1996.

[ADM98] O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java™ Virtual Machines. In *Proceedings of PLDI'98*, pages 269–279, Montreal, Canada, June 1998.

[AFG94] L. Amsaleg, M. J. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Workstation/Server Database Systems. Technical Report CS-TR-3370, Dept. of Computer Science, University of Maryland, College Park, MD, 1994.

[AFG95] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 42–54, Zurich, Switzerland, September 1995.

[AFG99] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In Atkinson and Welland [AW99], chapter 2.2.3, pages 427–430.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[Age98] O. Agesen. GC Points in a Threaded Environment. Technical Report TR-98-70, Sun Microsystems Laboratories, Palo Alto, CA, December 1998.

[AGO99] A. Albano, G. Ghelli, and R. Orsini. An Introduction to Fibonacci: a Programming Language for Object Databses. In Atkinson and Welland [AW99], chapter 1.1.2, pages 60–97.

[AJ99a] M. P. Atkinson and M. J. Jordan. Improved Hash Coding Methods for Java. Technical report, Sun Microsystems Laboratories, 1999. *In Preparation.*

[AJ99b] M. P. Atkinson and M. J. Jordan. Issues Raised by Three Years of Developing PJama: An Orthogonally Persistent Platform for Java™. In *Proceedings of ICDT'99*, Jerusalem, Israel, January 1999.

[AJ00] M. P. Atkinson and M. J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical report, Department of Computing Science, University of Glasgow, Scotland, 2000. *In Preparation.*

[AJDS96] 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 *Proceedings of POS'7*, Cape May, New Jersey, USA, May 1996.

[Ala97] S. Alagić. The ODMG Object Model: Does it Make Sense? In *Conference Proceedings of OOPSLA '97, Atlanta, GA, USA*, volume 32(10) of *ACM SIGPLAN Notices*, pages 253–270, October 1997.

[AM92] A. Albano and R. Morrison, editors. *Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Springer-Verlag.

[AM95] M. P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[And98] L. Anderson, GemStone Inc. Personal Communication, July 1998.

[AOV+99] M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. Zdonik, and M. Brodie, editors. *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*, Edinburgh, Scotland, September 1999. Morgan Kaufmann Publishers.

[App88] A. W. Appel. Simple Generational Garbage Collection and Stack Allocation. *Software — Practice and Experience*, 19(2):171–183, March 1988.

[Ard99a] E. Arderiu. Language Interoperability for a PetaByte Persistent Object Storage. In Chaudhri and Zimmermann [CZ99].

[Ard99b] E. Arderiu, CERN. Personal Communication, November 1999.

[ASL89] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A Query Language for Manipulating Object-Oriented Databases. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB'89)*, pages 433–442, Amsterdam, The Netherlands, August 1989.

[Atk78] M. P. Atkinson. Programming Languages and Databases. *VLDB Journal*, pages 408–419, 1978.

[Atk99] M. P. Atkinson. Personal Communication, November 1999.

[AW99] M. P. Atkinson and R. C. Welland, editors. *Fully Integrated Data Environments*. Springer-Verlag, 1999.

[AWO+99] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification*. Addison-Wesley, 1999.

[Bak78] H. G. Baker. List Processing in Real-Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[Bak95] H. G. Baker, editor. *Proceedings of the Second International Workshop on Memory Management*, number 986 in Lecture Notes in Computer Science, Kinross, Scotland, September 1995. Springer-Verlag.

[Bar88] J. F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, February 1988.

[Bar89] J. F. Bartlett. Mostly-Copying Garbage Collection Picks up Generations and C++. Technical Report TN-12, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, October 1989.

[BC92] Y. Bekkers and J. Cohen, editors. *Proceedings of the First International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St Malo, France, September 1992. Springer-Verlag.

[BDF+00] S. J. Baylor, M. Devarakonda, S. J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java Server Benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Implementing an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, 1992.

[BDN+98] S. M. Blackburn, L. Daynès, S. M. Nettles, D. Hulse, and O. J. Anfindsen. POS8 Keynote Discussion. Concurrency: The Fly in the Ointment. In Morrison et al. [MJA98], pages 215–222.

[Ber98] J. L. Bertoni. Understanding Solaris™ Filesystems and Paging. Technical Report TR-98-55, Sun Microsystems Laboratories, Palo Alto, CA, November 1998.

[BF98] X. Blondel and P. Ferreira. Implementing Garbage Collection in the PerDiS System. In Morrison et al. [MJA98], pages 64–77.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Conference Proceedings of OOPSLA '86, Portland, OR, USA*, volume 21(11) of *ACM SIGPLAN Notices*, pages 78–86, November 1986.

[Bis77] P. B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report MIT/LCS/TR-178, MIT Laboratory for Computer Science, MA, USA, 1977.

[BM92] A. L. Brown and R. Morrison. A Generic Persistent Object Store. *Software Engineering Journal*, pages 161–168, 1992. Also appears as FIDE Technical Report FIDE/92/39.

[BMM+99] A. L. Brown, G. Mainetto, F. Matthes, R. Müller, and D. McNally. An Open System Architecture for a Persistent Object Store. In Atkinson and Welland [AW99], chapter 2.2.1, pages 387–390.

[BNHC98] K. Brahnmath, N. Nystrom, A. L. Hosking, and Q. Cutts. Swizzle Barrier Optimizations for Orthogonal Persistence in Java. In Morrison et al. [MJA98], pages 268–278.

[BNOW93] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, pages 217–230, New York, NY, USA, December 1993. ACM Press.

[BR91] A. L. Brown and J. Rosenberg. Persistent Object Stores: An Implementation Technique. In *Proceedings of POS'4*. Morgan Kaufmann Publishers, 1991.

[Bra98] K. Brahnmath. Optimizing Orthogonal Persistence for Java. Master's thesis, Purdue University, IN, USA, May 1998.

[Bro89] A. L. Brown. *Persistent Object Stores*. PhD thesis, University of St Andrews, Scotland, October 1989.

[BW88] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience*, pages 807–820, September 1988.

[Byowww] J. Byous. Java™ Technology: An Early History. http://java.sun.com/features/1998/05/birthday.html [February 24, 2000].

[BZ98] S. M. Blackburn and J. N. Zigman. Concurrency — The Fly in the Ointment? In Morrison et al. [MJA98], pages 259–267.

[CADA87] R. 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 (VLDB'87)*, pages 117–125, Brighton, England, 1987.

[Cat97] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, 1997.

[CCM99] Q. Cutts, R. C. H. Connor, and R. Morrison. The PamCase Machine. In Atkinson and Welland [AW99], chapter 2.1.3, pages 346–364.

[CD87] M. J. Carey and D. J. DeWitt. An Overview of the EXODUS Project. *Database Engineering*, 10(2):107–114, 1987.

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):383–394, June 1994.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. J. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, September 1989. Revised.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, June 1993.

[CDRS86] M. J. Carey, D. J. DeWitt, J. Richardson, and E. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86)*, pages 1–27, Kyoto, Japan, 1986.

[CH97] Q. Cutts and A. L. Hosking. Analysing, Profiling and Optimising Orthogonal Persistence for Java. In Jordan and Atkinson [JA97]. Published as SunLabs Technical Report TR-97-63.

[Che70] C. J. Cheney. A Non-Recursive List Compacting Algorithm. *Communications of the ACM*, 11(13):677–678, November 1970.

[CKWZ96] E. J. Cook, A. W. Klauser, A. L. Wolf, and B. G. Zorn. Semi-Automatic, Self-Adaptive Control of Garbage Collection Rates in Object Databases. In *Proceedings of SIGMOD'96*, pages 377–388, Montreal, Canada, October 1996.

[CN96] R. C. H. Connor and S. Nettles, editors. *Persistent Object Systems: Principles and Practice — Proceedings of the Seventh International Workshop on Persistent Object Systems (POS7)*. Morgan Kaufmann Publishers, Cape May, NJ, USA, May 1996. Book was published in 1997.

[CNNwww] CNN. Astronauts Close Out Mapping Mission. http://www.cnn.com/2000/TECH/space/02/21/shuttle.01.ap/index.html [February 21, 2000].

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Coo97] R. Cooper. *Object Databases — an ODMG Approach.* International Thomson Computer Press, 1997.

[CSH+98] G. Clossman, P. Shaw, M. Hapner, J. Klein, R. Pledereder, and B. Becker. Java and Relational Databases: SQLJ. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2), June 1998.

[CWZ94] J. E. Cook, A. L. Wolf, and B. G. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proceedings of SIGMOD'94*, pages 371–382, Minneapolis, USA, May 1994.

[CZ99] A. B. Chaudhri and J. Zimmermann, editors. *OOPSLA'99 Workshop on Java™ and Databases: Persistence Options*, Denver, CO, USA, November 1999.

[DA97] L. Daynès and M. P. Atkinson. Main-Memory Management to support Orthogonal Persistence for Java. In Jordan and Atkinson [JA97]. Published as SunLabs Technical Report TR-97-63.

[DA99a] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *Proceedings of ECOOP'99*, pages 258–278, Lisbon, Portugal, June 1999.

[DA99b] M. Dmitriev and M. P. Atkinson. Evolutionary Data Conversion in the PJama Persistent Language. In *Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases*, Lisbon, Portugal, June 1999.

[DAV96] L. Daynès, M. P. Atkinson, and P. Valduriez. Efficient Support for Customizing Concurrency Control in Persistent Java. In *Proceedings of the International Workshop on Advanced Transaction Models and Architectures*, Goa, India, August 1996.

[DAV97] L. Daynès, M. P. Atkinson, and P. Valduriez. Customizable Concurrency Control for Persistent Java. In Jajodia and Kerschberg [JK97], chapter 7.

[Day96] L. Daynès. Extensible Transaction Management in PJava. In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[DdF+94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindstrom, J. Rosenberg, and F. Vaughan. Grasshopper: an Orthogonally Persistence Operating System. *Computing Systems*, 7(3):289–312, Summer 1994.

[Det90] D. Detlefs. Concurrent, Atomic Garbage Collection. In *Workshop on Garbage Collection*, October 1990.

[Det91] D. Detlefs. *Concurrent, Atomic Garbage Collection.* PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, November 1991.

[DHF96] A. Dearle, D. Hulse, and A. Farkas. Operating System Support for Java. In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[Dmi98] M. Dmitriev. The First Experience of Class Evolution Support in PJama. In Morrison et al. [MJA98], pages 279–296.

[Dun99] I. Dunham *et al.* The DNA Sequence Of Human Chromosome 22. *Nature*, 402:489–495, December 1999.

[ELA88] J. R. Ellis, K. Li, and A. W. Appel. Real-Time Concurrent Collection on Stock Multiprocessors. Technical Report 25, Systems Research Center, Digital Equipment Corporation, February 1988.

[EMS91] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon — A Distributed Transaction Facility.* Morgan Kaufmann Publishers, 1991.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin Cummings, 1994. Second Edition.

[Engwww] J. English. The Story of the Java™ Platform. http://java.sun.com/nav/whatis/storyofjava.html [February 24, 2000].

[Eva00] H. Evans. Why Object Serialization is Inappropriate for Providing Persistence in Java. Technical report, Department of Computing Science, University of Glasgow, Scotland, 2000. *In Preparation.*

[FS95] P. Ferreira and M. Shapiro. Garbage Collection in the Larchant Persistent Distributed Shared Store. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'95)*, pages 461–467, August 1995.

[FS96] P. Ferreira and M. Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In *Sixteenth International Conference on Distributed Computer Systems (ICDCS'96)*, Hong Kong, May 1996.

[Fujwww] Fujitsu Ltd. 3.5-inch Magnetic Disk Drives MAB3045/MAB3091. http://www.fujitsu.co.jp/hypertext/hdd/drive/overseas/mab30xx/mab30xx.html [January 5, 2000].

[FvFH93] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics — Principles and Practice.* Addison-Wesley, November 1993. Second Edition.

[FW85] W. Findlay and D. A. Watt. *Pascal.* UCL Press, December 1985.

[FZT+92] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey, and D. J. DeWitt. Crash recovery in client server EXODUS. In *Proceedings of SIGMOD'92*, pages 165–174, July 1992.

[Gar99] A. Garthwaite, Sun Microsystems Inc. Personal Communication, November 1999.

[Gem98a] GemStone Systems Inc. *GemStone/J™ Administration for Unix*, March 1998. Version 1.1.

[Gem98b] GemStone Systems Inc. *GemStone/J™ Programming Guide*, March 1998. Version 1.1.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[Goswww] J. Gosling. A Brief History of the Green Project. http://java.sun.com/people/jag/green/index.html [February 24, 2000].

[Gos93] J. Gosling. *Oak Language Specification*, 1993. Version 2.0.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[GSAW98] S. Grimstad, D. I. K. Søberg, M. P. Atkinson, and R. C. Welland. Evaluating Usability Aspects of PJama based on Source Code Measurements. In Morrison et al. [MJA98], pages 307–321.

[Gut84] A. Guttman. R-Trees: a Dynamic Index Structure for Spatial Searching. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 14(2):47–57, 1984.

[HAD99] 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, Scotland, December 1999.

[Ham97] C. G. Hamilton. Measuring the Performance of Disk Garbage Collectors: Garbage Collecting Persistent Java Stores. Master's thesis, Department of Computing Science, University of Glasgow, Scotland, 1997.

[Ham99a] C. G. Hamilton. Recovery Management for Sphere: Recovering A Persistent Object Store. Technical Report TR-1999-51, Department of Computing Science, University of Glasgow, Scotland, December 1999.

[Ham99b] J. Hamilton. Networked Data Management Design Points. In Atkinson et al. [AOV+99], pages 202–206.

[Ham00] C. G. Hamilton. Log Management for Sphere: Logging A Persistent Object Store. Technical report, Department of Computing Science, University of Glasgow, Scotland, 2000. *In Preparation.*

[HBM93] A. L. Hosking, E. Brown, and J. E. B. Moss. Update Logging for Persistent Programming Languages: A Comparative Performance Evaluation. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB'93)*, pages 429–440, Dublin, Ireland, August 1993. Morgan Kaufmann Publishers.

[HC99a] A. L. Hosking and J. Chen. Mostly-Copying Reachability-Based Orthogonal Persistence. In OOPSLA'99 [OOP99], pages 382–398.

[HC99b] A. L. Hosking and J. Chen. PM3: An Orthogonally Persistent Systems Programming Language — Design, Implementation, Performance. In Atkinson et al. [AOV+99], pages 587–598.

[HM92] R. L. Hudson and J. E. B. Moss. Incremental Garbage Collection of Mature Objects. In Bekkers and Cohen [BC92], pages 388–403.

[HM93a] A. L. Hosking and J. E. B. Moss. Object Fault Handling for Persistent Programming Languages: A Performance Evaluation. In *Proceedings of OOPSLA'93*, pages 288–303, Washington, DC, USA, October 1993.

[HM93b] A. L. Hosking and J. E. B. Moss. Protection Traps and Alternatives for Memory Management of an Object-Oriented Language. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, pages 106–119, New York, NY, USA, December 1993. ACM Press.

[HMB90] A. L. Hosking, J. E. B. Moss, and C. Bliss. Design of an Object Faulting Persistent Smalltalk. Technical Report UM-CS-1990-45, Department of Computer and Information Science, University of Massachussets, MA, USA, May 1990.

[HMMM97] R. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage Collecting the World: One Car at a Time. In OOPSLA'97 [OOP97], pages 162–175.

[HNCB98] A. L. Hosking, N. Nystrom, Q. Cutts, and K. Brahnmath. Optimizing the Read and Write Barriers for Orthogonal Persistence. In Morrison et al. [MJA98], pages 149–159.

[Hos96] A. L. Hosking. Residency Check Elimination for Object-Oriented Persistent Languages. In Connor and Nettles [CN96], pages 174–183. Book was published in 1997.

[Hos99] A. L. Hosking, Purdue University. Personal Communication, September 1999.

[How98] J. Howell. Straightforward Java Persistence through Checkpointing. In Morrison et al. [MJA98], pages 322–333.

[Ing86] D. H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of OOPSLA'86*, Portland, Oregon, USA, 1986.

[ISO98] ISO. Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++, 1998. ISO/IEC 14882-1998.

[JA96] M. J. Jordan and M. P. Atkinson, editors. *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland, November 1996. Published as SunLabs Technical Report TR-96-58.

[JA97] M. J. Jordan and M. P. Atkinson, editors. *Proceedings of the Second International Workshop on Persistence and Java*, Half Moon Bay, CA, USA, December 1997. Published as SunLabs Technical Report TR-97-63.

[JA98] M. J. Jordan and M. P. Atkinson. Orthogonal Persistence for Java — A Mid-term Report. In Morrison et al. [MJA98], pages 335–352.

[JA99] M. J. Jordan and M. P. Atkinson, editors. Orthogonal Persistence for the Java™ Platform — Draft Specification. Technical report, Sun Microsystems Inc, 1999.

[JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures.* Kluwer Academic Publishers, August 1997.

[Joh97] M. S. Johnstone. Personal Communication, October 1997.

[Jon96] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[Jon98] R. E. Jones, editor. *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, Canada, October 1998. ACM Press.

[Jor96a] M. J. Jordan. Early Experiences with Persistent Java. In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[Jor96b] M. J. Jordan, Sun Microsystems Inc. Personal Communication, Summer 1996.

[Jor99a] D. Jordan. Serialisation is not a database substitute. *Java™ Report*, pages 68–79, July 1999.

[Jor99b] M. J. Jordan. The Java Platform as a Database. In Chaudhri and Zimmermann [CZ99].

[JV95] 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, USA, September 1995.

[JV97] M. J. Jordan and M. L. Van De Vanter. Modular System Building with Java Packages. In *Eighth International Conference on Software Engineering Environments*, pages 155–163, Cottbus, Germany, May 1997.

[JW98] M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? In Jones [Jon98].

[Kak98] S. V. Kakkad. *Address Translation and Storage Management for Persistent Object Stores.* PhD thesis, University of Texas, Austin, TX, USA, March 1998. Technical Report CS-TR-98-07.

[KCC+97] G. N. C. Kirby, R. C. H. Connor, Q. I. Cutts, R. Morrison, D. S. Munro, and S. Scheuerl. Flask: An Architecture Supporting Concurrent Distributed Persistent Applications". Technical Report CS/97/4, University of St Andrews, Scotland, 1997.

[KJW98] S. V. Kakkad, M. S. Johnstone, and P. R. Wilson. Portable Run-Time Type Description for Conventional Compilers. In Jones [Jon98], pages 146–153.

[KL92] D. C. Kay and J. R. Levine. *Graphics File Formats*. Windcrest/McGraw Hill, 1992.

[KLW89] E. K. Kolodner, B. Liskov, and W. E. Weihl. Atomic Garbage Collection: Managing a Stable Heap. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 18(2):15–25, June 1989.

[Kno65] K. C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–625, October 1965.

[Kol87] E. K. Kolodner. Recovery Using Virtual Memory. Technical Report MIT/LCS/TR-404, MIT Laboratory for Computer Science, MA, USA, July 1987.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Software Series, 1988. ANSI C, Second Edition.

[KW93] E. K. Kolodner and W. E. Weihl. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):177–186, June 1993.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. J. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Quebec, Canada, June 1996.

[LB98] B. Lewis and D. J. Berg. *Multithreaded Programming with PThreads*. Sun Microsystems Press, A Prentice Hall Title, 1998.

[LCD+94] B. Liskov, D. Curtis, M. Day, S. Ghemawhat, R. Gruber, P. Johnson, and A. C. Myers. *Theta Reference Manual*, February 1994.

[Lew00] B. Lewis, Sun Microsystems Inc. Personal Communication, January 2000.

[LH83] H. Lieberman and C. E. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[Lie93] J. Liedtke. A Persistent System in Real Use: Experiences of the First 13 Years. In *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, December 1993.

[Lie96] J. Liedtke. *L4 Reference Manual*, September 1996.

[Lis84] B. Liskov. Overview of the Argus Language and System. Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, MA, USA, February 1984.

[LLB+97] G. Landis, C. Lamb, T. Blackman, S. Haradhvala, M. Noyes, and D. Weinreb. ObjectStore PSE: a Persistent Storage Engine for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997.

[LLOW91] C. Lamb, G. Landis, J. Orestein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[LM99] 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"*, Denver, Colorado, USA, November 1999.

[LY99] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification — Second Edition.* Addison-Wesley, 1999. Java™ 2 Platform.

[Mah97] U. Maheshwari. *Garbage Collection in a Large, Distributed, Object Store.* PhD thesis, MIT Laboratory for Computer Science, MA, USA, September 1997. Technical Report MIT/LCS/TR-727.

[Mat99] F. Matthes. Higher-Order Persistent Polymorphic Programming in Tycoon. In Atkinson and Welland [AW99], chapter 1.1.1, pages 13–59.

[MBMM98] D. S. Munro, A. L. Brown, R. Morrison, and J. E. B. Moss. Incremental Garbage Collection of a Persistent Object Store using PMOS. In Morrison et al. [MJA98], pages 78–91.

[MCC+99] R. Morrison, R. C. H. Connor, Q. Cutts, G. N. C. Kirby, D. S. Munro, and M. P. Atkinson. The Napier88 Persistent Programming Language and Environment. In Atkinson and Welland [AW99], chapter 1.1.3, pages 98–154.

[MCKM96] R. Morrison, R. C. H. Connor, G. N. C. Kirby, and D. S. Munro. Can Java Persist? In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[MCM+94] D. S. Munro, R. C. H. Connor, R. Morrison, S. Scheuerl, and D. Stemple. Concurrent Shadow Paging in the Flask Architecture. In *Proceedings of the Sixth International Workshop on Persistent Object Systems*, pages 16–42, Tarascon, France, 1994. Springer-Verlag.

[MH96] J. E. B. Moss and A. L. Hosking. Approaches to Adding Persistence to Java. In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[MHL+89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. Technical report, IBM Research Division, Almaden Research Center, January 1989.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwartz. 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.

[MJA98] 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 Publishers, Tiburon, CA, USA, August 1998.

[ML97] U. Maheshwari and B. Liskov. Partitioned Garbage Collection of Large Object Store. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 313–323, Tucson, Arizona, June 1997.

[MMH96] J. E. B. Moss, D. S. Munro, and R. L. Hudson. PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores. In Connor and Nettles [CN96]. Book was published in 1997.

[MMS94] F. Matthes, S. Müßig, and J. W. Schmidt. Persistent Polymorphic Programming in Tycoon: an Introduction. Technical report, Fachbereich Informatik, Universität Hamburg, Germany, 1994.

[MMS99] F. Matthes, R. Müller, and J. W. Schmidt. Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol. In Atkinson and Welland [AW99], chapter 2.2.4, pages 431–433.

[Moh99] C. Mohan. Repeating History Beyond ARIES. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*, pages 2–17, Edinburgh, Scotland, September 1999. Morgan Kaufmann Publishers. Ten-Year Award Paper.

[Mor79] R. Morrison. S-algol Language Reference Manual. Technical Report CS/79/1, University of St Andrews, 1979.

[Mos90a] J. E. B. Moss. Design of the Mneme Persistent Object Store. Technical report, Department of Computer and Information Science, University of Massachussets, MA, USA, August 1990.

[Mos90b] J. E. B. Moss. Working With Objects: To Swizzle or Not to Swizzle? Technical Report UM-CS-1990-038, Department of Computer and Information Science, University of Massachussets, MA, USA, July 1990.

[MS93] J. Melton and A. R. Simon. *Understanding the new SQL: a Complete Guide.* Morgan Kaufmann Publishers, 1993.

[MS94] H. H. Mashburn and M. Satyanarayanan. *RVM: Recoverable Virtual Memory*, June 1994.

[MSS99] F. Matthes, G. Schröder, and J. W. Schmidt. Tycoon: A Scalable and Interoperable Persistent System Environment. In Atkinson and Welland [AW99], chapter 2.1.4, pages 365–381.

[Mun93] D. S. Munro. *On the Integration of Concurrency, Distribution and Persistence.* PhD thesis, University of St Andrews, Scotland, 1993.

[Mun99] R. Munz, Program Director DBMS-Technology, SAP AG. Usage Scenarios for DBMS, September 1999. Keynote Address at VLDB'99, Edinburgh, Scotland.

[MV99] T. Murer and M. L. Van De Vanter. Replacing Copies With Connections: Managing Software across the Virtual Organization. In *Proceedings of the 8th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Stanford University, CA, USA, June 1999.

[NASwww] NASA. Shuttle Mission STS-99. http://www.ksc.nasa.gov/shuttle/missions/sts-99/mission-sts-99.html [February 21, 2000].

[Natwww] National Center for Biotechnology Information. Human Genome Sequencing. http://www.ncbi.nlm.nih.gov/genome/seq/ [December 7, 1999].

[NHW+99] N. Nystrom, A. L. Hosking, D. Whitlock, Q. Cutts, and A. Diwan. Partial Redundancy Elimination for Access Path Expressions. In *Proceedings of the International Workshop on Aliasing in Object-Oriented Stystems*, Lisbon, Portugal, June 1999.

[NOG93] S. Nettles, J. O'Toole, and D. Gifford. Concurrent Garbage Collection of Persistent Heaps. Technical Report CS-93-137, School of Computer Science, Carnegie Mellon University, April 1993.

[NOPH93] S. Nettles, J. O'Toole, D. Pierce, and N. Haines. Replication-Based Incremental Copying Collection. Technical Report CS-93-135, School of Computer Science, Carnegie Mellon University, April 1993.

[Nys98] N. Nystrom. Bytecode Level Analysis and Optimization of Java Classes. Master's thesis, Purdue University, USA, August 1998.

[Objwwwa] Objectivity Inc. Objectivity for Java.
http://www.objectivity.com/Products/prodov.html#Java [February 23, 2000].

[Objwwwb] ObjectSpace. JGL v3.1.
http://www.objectspace.com/products/prodJGL.asp [February 23, 2000].

[Obj99] ObjectDesign Inc. *ObjectStore PSE/PSE Pro for Java API User Guide* , November 1999.

[OLC97] J. Oler, G. Lindstrom, and T. Critchlow. Migrating Relational Data to an ODBMS: Strategies and Lessons FRom a Molecular Biology Experience. In OOPSLA'97 [OOP97], pages 243–252.

[ON93] J. O'Toole and S. Nettles. Concurrent Replication Garbage Collection: An Implementation Report. Technical Report CMU-CS-93-138, School of Computer Science, Carnegie Mellon University, April 1993.

[ON94] J. O'Toole and S. Nettles. Concurrent Replicating Garbage Collection. In *Conference on Lisp and Functional programming*. ACM Press, June 1994.

[OOP97] *OOPSLA'97, ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 32(10) of *ACM SIGPLAN Notices*, Atlanta, GA, USA, October 1997. ACM Press.

[OOP99] *OOPSLA'99, ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, USA, November 1999. ACM Press.

[Orawww] Oracle Corp. Oracle and Java.
http://www.oracle.com/java/ [February 23, 2000].

[Oti96] A. Otis. Persistent Java Issues From a Gemstone Perspective. In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[Oti98] A. Otis, GemStone Inc. Personal Communication, Summer 1998.

[Ous90] J. K. Ousterhout. Tcl: an Embeddable Command Language. In *Proceedings of the USENIX Association Winter Conference*, winter 1990.

[Ous93] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1993.

[PA00] T. Printezis and M. P. Atkinson. An Efficient Object Promotion Algorithm for Persistent Object Systems, 2000. Accepted for publication at *Software – Practice and Experience.*

[PAD+97a] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. J. Bailey. The Design of a new Persistent Object Store for PJama. In Jordan and Atkinson [JA97]. Published as SunLabs Technical Report TR-97-63.

[PAD+97b] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. J. 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, Scotland, August 1997.

[PAD98] 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, Scotland, May 1998.

[PAJ99] 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. *To be published.*

[PC96] T. Printezis and Q. Cutts. Measuring the Allocation Rate of Napier88. Technical Report TR-1996-44, Department of Computing Science, University of Glasgow, Scotland, November 1996.

[PD00] T. Printezis and D. Detlefs. A Generational Mostly-Concurrent Garbage Collector, 2000. To be submitted to the *International Symposium on Memory Management (ISMM 2000).*

[PHA+96] J. Peterson, K. Hammond, L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. D. Gordon, J. Hughes, T. Johnsson, M. Jones, S. Peyton Jones, A. Reid, and P. Wadler. *Report on the Programming language Haskell (version 1.3)*, May 1996.

[Phi99] G. Phipps. Comparing Observed Bug and Productivity Rates for Java and C++. *Software — Practice and Experience*, 29(4):345–358, April 1999.

[PN77] J. L. Peterson and T. A. Norman. Buddy Systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[POEwww] POET Software. POET Object Server Suite 6.0. http://www.poet.com/products/oss/oss.html [February 23, 2000].

[Pri96] T. Printezis. Analysing a Simple Disk Garbage Collector. In Jordan and Atkinson [JA96]. Published as SunLabs Technical Report TR-96-58.

[Pri99a] T. Printezis. Orthogonal Persistence: The Future for Storing Objects? In *Proceedings of the Practical Applications for Java Conference 1999 (PAJava'99)*, pages 5–17, London, UK, April 1999. *Invited Paper.*

[Pri99b] T. Printezis. The Sphere User's Guide. Technical Report TR-1999-47, Department of Computing Science, University of Glasgow, Scotland, July 1999.

[PS95] D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In Baker [Bak95].

[RCS93] 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), July 1993.

[RLJ98] D. Raggett, A. Le Hors, and I. Jacobs, editors. HTML 4.0 Specification, April 1998. World Wide Web Consortium (W3C) Recommendation.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB'89)*, pages 337–346, Amsterdam, The Netherlands, August 1989. Morgan Kaufmann Publishers.

[RTW97] J. V. E. Ridgway, C. Thrall, and J. C. Wileden. Towards Assessing Approaches to Persistence for Java. In Jordan and Atkinson [JA97]. Published as SunLabs Technical Report TR-97-63.

[SA97] S. Spence and M. P. Atkinson. A Scalable Model of Distribution Promoting Autonomy of and Cooperation Between PJava Object Stores. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Hawaii, USA, January 1997.

[SBM96] M. Stonebraker, P. Brown, and D. Moore. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publishers, San Francisco, 1996.

[Sch77] J. W. Schmidt. Some High Level Language Constructs for Data of Type Relation. In *Proceedings of SIGMOD'77*, August 1977.

[Sed88] R. Sedgewick. *Algorithms.* Addison-Wesley, 1988. Second Edition.

[SF95] M. Shapiro and P. Ferreira. Larchant–RDOSS: a Distributed Shared Persistent Memory and its Garbage Collector. Technical Report BROADCAST#TR95-88, INRIA, Rocquencourt, June 1995.

[SG95] J. Seligmann and S. Grarup. Incremental Mature Garbage Collection using the Train Algorithm. In *European Conference on Object-Oriented Programming*. Springer-Verlag, August 1995.

[Sku97] M. Skubiszewski. Personal Communication, August 1997.

[SKW92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In Albano and Morrison [AM92], pages 11–33.

[SP93] P. Sansom and S. L. Peyton Jones. Generational Garbage Collection for Haskell. In *Functional Programming and Computer Architecture*, June 1993.

[SP96] M. Skubiszewski and N. Porteix. GC-consistent Cuts of Databases. Technical Report 2681, INRIA, Rocquencourt, April 1996.

[SP97] M. Skubiszewski and N. Porteix. Partly-Consistent Cuts of Databases. *Lecture Notes in Computer Science*, 1300, 1997.

[SPA94] SPARC International Inc. *SPARC Architecture Manual Version 9.* Prentice Hall, 1994.

[Spe98] S. Spence. *Persistent RMI.* Department of Computing Science, University of Glasgow, Scotland, March 1998.

[Spe99] S. Spence. PJRMI: Remote Method Invocation for a Persistent System. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, September 1999. IEEE Press.

[Sunwww] Sun Microsystems Inc. Workgroup Servers, Sun Enterprise™ 450. http://www.sun.com/servers/workgroup/450/ [January 5, 2000].

[Sun89] Sun Microsystems Inc. NFS: Network File System Protocol Specification. RFC 1094, Network Information Center, SRI International, March 1989.

[Sun97] Sun Microsystems Inc. *JavaBeans™*, July 1997. Version 1.01.

[Sun98a] Sun Microsystems Inc. *JavaSpaces™ Specification*, July 1998. Revision 1.0 Beta.

[Sun98b] Sun Microsystems Inc. *Java™ Object Serialization Specification — JDK™ 1.2*, November 1998. Revision 1.43.

[Sun98c] Sun Microsystems Inc. *Java™ Platform 1.2 Core API Specification*, 1998.

[Sun98d] Sun Microsystems Inc. *Java™ Remote Method Invocation Specification*, October 1998. Revision 1.5.

[Sun98e] Sun Microsystems Inc. *JDBC™ 2.0 API*, May 1998. Version 1.0.

[Sun98f] Sun Microsystems Inc and The University of Glasgow. *PJama API*, 1998. Release 0.5.7.13.

[Sun99a] Sun Microsystems Inc. *Java 3D™ API Specification*, August 1999. Version 1.2_alpha1.

[Sun99b] Sun Microsystems Inc. *Programmer's Guide to the Java™ 2D API — Enhanced Graphics and Imaging for Java*, May 1999. Java™ 2 SDK, Standard Edition, 1.2 Version.

[Sup98a] SuperCede Inc. *A Reference Manual for the Application Programming Interface from the Java language to SuperCede™ PersistentMemory™*, 1998.

[Sup98b] SuperCede Inc. *SuperCede™ for Java — User Guide*, 1998.

[SV97] M. Skubiszewski and P. Valduriez. Concurrent Garbage Collection in O_2. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 356–365, Athens, Greece, August 1997.

[Tai98] A. Taivalsaari. Implementing a Java™ Virtual Machine in the Java Programming Language. Technical Report TR-98-64, Sun Microsystems Laboratories, Palo Alto, CA, March 1998.

[Tan87] A. S. Tanenbaum. *Operating Systems — Design and Implementation.* Prentice Hall International Editions, 1987.

[Tan90] A. S. Tanenbaum. *Structured Computer Organisation.* Prentice Hall International Editions, 1990. Third Edition.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall International Editions, 1992.

[Thewww] The PJama Project. Orthogonal Persistence for Java, PJama Release 1.6.4 (for JDK 1.2). http://www.sun.com/research/forest/opj.main.html [March 10, 2000].

[TLA90] A. M. Tenenbaum, Y. Langsam, and M. J. Augenstein. *Data Structures Using C*. Prentice Hall International Editions, 1990.

[Ung84] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[US www] US Census Bureau. TIGER/Line® Web Pages. http://www.census.gov/geo/www/tiger/index.html/ [December 8, 1999].

[US 98] US Census Bureau. *TIGER/Line® Files 1998, Technical Documentation*, July 1998.

[Van98] M. L. Van De Vanter. Coordinated Editing of Versioned Packages in the JP Programming Environment. In *Proceedings of the 8th Symposium on System Configuration Management (SCM-8)*, Brussels, Belgium, July 1998.

[Verwww] Versant Corp. J/Versant Interface (JVI) Release 2. http://www.versant.com/us/products/java/index.html [February 23, 2000].

[VM199] M. L. Van De Vanter and T. Murer. Global Names: Support for Managing Software in a World of Virtual Organizations. In *Proceedings of the 9th Symposium on System Configuration Management (SCM-9)*, Toulouse, France, September 1999.

[Wal99] J. Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[WD92] S. J. White and D. J. DeWitt. A Performance Study of Alternative Object Faulting and Point Swizzling Strategies. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)*, pages 419–431, San Mateo, CA, USA, August 1992. Morgan Kaufmann Publishers.

[WG99] D. White and A. Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories, 1999.

[Wil92] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In Bekkers and Cohen [BC92], pages 1–42.

[Wil97] P. R. Wilson. Personal Communication, August 1997.

[Wis78] D. S. Wise. The Double Buddy-System. Technical Report 79, Computer Science Department, Indiana University, Bloomington, Indiana, December 1978.

[WJNB95] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In Baker [Bak95].

[WK92] 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 International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.

[YNY94] V. F. Yong, J. F. Naughton, and J. B. Yu. Storage Reclamation and Reorganization in Client-Server Persistent Object Stores. In *Proceedings of the International Conference on Data Engineering*, 1994.

[Zor92] B. G. Zorn. Barrier Methods for Garbage Collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, August 1992.

[Zor93] B. G. Zorn. The Measured Cost of Conservative Garbage Collection. *Software — Practice and Experience*, 23(7):733–756, 1993.