The Australian Business Register Demonstrator: Addressing Complexity and Scale in a High Performance Object Server.

Alonso Marquez and Stephen M Blackburn

Abstract

The so called 'information explosion' has lead to an enormous demand for networked information, which has in turn placed enormous pressure on information servers. The design of high performance server technology has thus become a hot topic in computer science. In this paper we describe how our exposure to a large real-world application has shaped our response to two key issues for high performance object server technologies—the management of complexity and the management of scale.

The Australian Bureau of Statistics' Business Register (BR) is an OODB application that forms a register of all businesses in Australia. The register is fundamental to much of the ABS' economic survey activity and is used by many branches of the large organization. The basis of the register is an object model that reflects the business structure of all Australian businesses, from large multinationals through to corner stores. There are over 100,000,000 objects in the register and it is constantly updated, both through operator-driven data entry and the batch processing of data from other government agencies. Specific requirements are acceptable response time for interactive access for more that 100 users, long and short transaction support, efficient storage and online access to historic information, flexible object version views and schema evolution support.

The challenges raised by the BR are dominated by the problems of complexity and scale. Our response to these challenges has been guided by our view that *abstraction* has a basic role in addressing complexity and scale, and by our use of *Java* as an implementation context.

In this paper we report not only our approach, but also novel technological outcomes which stem from our exposure to the challenges highlighted by the ABS-BR. These outcomes include: portable orthogonally persistent Java, a system for object versioning, a proposal for schema versioning, and a system architecture for scalable object servers based on a separation of programming language and storage system concerns. These outcomes are examined in the context of an ABS-BR technology demonstrator—a small-scale version of the ABS-BR built to test and showcase new technologies for high performance object servers.

1 Introduction

The communications revolution has made possible the explosive growth in the internet currently being witnessed. This growth has lead to a corresponding explosion in demand for networked information and has thrown server technology under the spotlight—its capacity for scalability and robustness now the subject of great scrutiny. While relational technology has an important role to play in some application domains, there are others where object technology is better suited. So it is that the high performance object server technology has become extremely important, any advance in the utility and scalability of such systems being of direct relevance to a large and rapidly growing market.

It was in this setting that we began a project with the Australian Bureau of Statistics (ABS) to examine their Business Register (BR) application with a view to exposing us to a large 'real world' problem, and allowing the ABS to see something of what the research community was doing to address the sort of problems they were facing. Through discussions with programmers and managers from this large project we were able to get a view 'from the coal-face' of some of the major problems faced today by implementers of large-scale object server applications. We were thus provided at once with both a source of motivating problems and a testing ground for our ideas.

Our exposure to the ABS-BR led to the identification of two major barriers: *complexity* and *scalability*. We approached the problems that these raised through a two-level strategy which involved the use of *abstraction* at a high level and the use of *Java* at an implementation level. Abstraction serves to separate concerns and reduce complexity, both of which aid scalability. Our choice of Java as an implementation environment was based on many of its popular features, but above all, on its capacity to be semantically extended—a feature that allowed us to efficiently and portably realize our strategy of complexity reduction through abstraction.

Part of our goal in working with the ABS was to provide us with a context for testing and analyzing our ideas. It was in this light that we built the ABS-BR demonstrator. The demonstrator implements a cut-down version of the ABS-BR over synthetic data[1], using orthogonally persistent Java (OPJ) with orthogonal versioning. These extensions to Java were developed by our research group and utilized underlying stores built to the PSI [6] storage interface by the group. The outcomes of this major implementation experiment is reported in some detail later in this paper. Suffice it to say here that through the abstractions provided by orthogonal persistence and orthogonal versioning, our implementation was enormously simplified. As to whether our objectives with respect to scalability were met, it is harder to say as we were not able to replicate the full ABS-BR for a variety of reasons.

The remainder of this paper is structured as follows. In section 2 the ABS-BR application is described and key challenges that it raises are discussed. Section 3 outlines our approach to the challenges of complexity and scale in terms of a high level strategy based on the use of abstraction and an implementation strategy based on the use of Java. This is followed in section 4 with a discussion of four key technologies used to address the ABS-BR challenges: orthogonally persistent Java, orthogonal versioning, the semantic extension framework for Java, and PSI, a persistent storage interface. Section 4 concludes with a description of our ABS-BR demonstrator which demonstrates the use of the aforementioned technologies in a scaled-down implementation of the ABS-BR. In section 5 we analyze the performance of the ABS-BR demonstrator and discuss the implications of the lessons of the ABS-BR, particularly in the context of the ODMG standard [5, 12].

2 The ABS Business Register

The ABS Business Register (BR) is a register of all businesses in Australia, from transnationals to corner stores. The BR is currently implemented using OODBMS technology, each business being modeled according to a complex schema that reflects real-world business structure, including concepts such as legal entities, enterprise groups, locations, etc. Key issues for the BR include *scalability*, *complexity*, *reliability*, *change management*, and *heterogeneity* of computational environment.

Scalability concerns include: storage—the database currently has over 100,000,000 objects using more than 10GB of storage space; processing capacity—the system must be able to respond to queries in a timely manner and be able to batch-process large quantities of data from external sources; and external IO—a large number of external clients must be supported simultaneously.

The complexity of the database is evidenced by a large schema and the need to preserve information about complex statistical data dependencies. This complexity is exacerbated by the difficulty of maintaining persistent-to-transient data mappings, and the demands of explicit storage management (alloc/free). As a central resource for the ABS's economic statistics, the reliability of the system is of paramount importance from the perspective of both data integrity and availability.

Change management places considerable demands on the system at a number of levels: the capacity to deal with long transactions (update transactions on large companies can take many days by virtue of the complexity of the data entry task); the need for object instance versioning to allow for retrospective queries and changes; and the need for schema versioning in order to support schema change and retrospective use of the data and associated schema.

The BR must do all of this in support of a wide range of users, including system operators, data entry personnel, and statisticians who exist within a disperse heterogeneous computing environment.

2.1 Key Issues Raised by the ABS-BR

The challenges of the ABS-BR and the difficulties faced by the ABS in realizing and maintaining a solution indicate a number of issues that we see as being of broad relevance to high performance object server technology. It is these issues that have been the focus of our research effort centered around the ABS-BR. Of these, the first four relate to system requirements that led to significant complexity for the ABS-BR application programmers, while the last relates to scalability. The challenges of the ABS-BR can thus be broadly classified as those of *complexity* and *scalability*.

Figure 1 illustrates some of the important concepts in the ABS-BR. *Historic snapshots* are globally consistent snapshots of the BR which define the resolution at which historical queries are made. The *common frame* is the current stable view of the BR (typically corresponding to the last historic snapshot). The *current view* is the view of the database that includes all updates since the last historic snapshot. *Dependent source feedback* refers to updates that in some way are a function of the database itself (e.g. where change in the database triggers the collection of new

[1] Strict privacy legislation prevents us using live ABS data. We were therefore restricted to the use of a synthetic facsimile of the Business Register data.

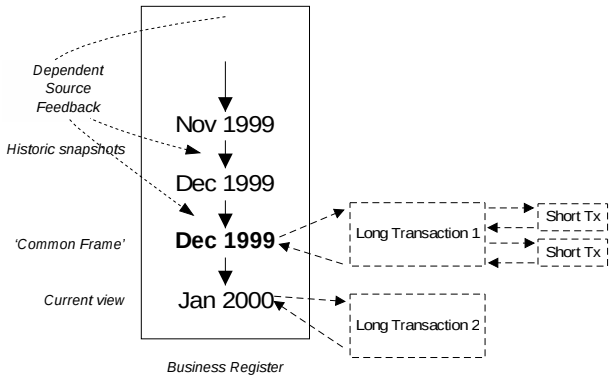

Figure 1: The ABS-BR showing historic snapshots, a 'common frame', long and short transactions, and an example of dependent source feedback.

data items for the database). Such data items constitute a feedback loop and therefore must be excluded from certain statistical analyses. Long and short transactions are depicted with respect to different views of the BR.

2.1.1 Need to Reduce Impedance Mismatch

Many OODBMS systems expose application programmers to the existence of transient (in-memory) and persistent (database) copies of the same data. In some cases the OODBMS only maintains the persistent copy, leaving the programmer with the task of managing mappings between, and coherency of persistent and transient copies. While the transient copy is used for implementing and realizing the application *logic*, the persistent copy must be maintained in order to ensure that changes are made stable, that transactional semantics are observed, and to facilitate inter-transactional data sharing. This adds substantial complexity to the programmer's task, sometimes doubling the size of the schema (with transient and persistent analogues of each class). In the case of the ABS-BR, this issue was a *major* cause of implementation complexity. An environment that allowed the programmer to abstract over the issue of persistent storage management would greatly simplify the task of the application programmer.

2.1.2 Need for Automatic Storage Management

Discussions with ABS-BR's application programmers revealed that storage management (alloc/free), particularly with respect to persistent data, was a major cause of complexity. Indeed, the complexity was so great that the problem of storage reclamation was deferred, with the consequence that the database grew monotonically at a rate in the order of 1 GB/month. The additional cost of disk storage was deemed insignificant when compared with the programming effort needed to (safely) address the intricate and complex problem. Unsurprisingly, the possibility of support for automatic storage management (through the use of garbage collection) was greeted with enthusiasm by the ABS-BR application programmers we spoke to.

2.1.3 Need to Support Historical Versioning

An important requirement of the ABS-BR is that it support historical queries. Furthermore, additional statistical requirements such as the need to avoid dependent source feedback add to the complexity of historical queries. In the context of these requirements, OODBMS support for versioning was deemed inadequate, and versioning support was built in explicitly at the application level, substantially adding to the complexity of the application. Not only must the application manage persistent and transient copies of objects (section 2.1.1), but multiple versions of the object, each with its own persistent and transient copies. All of these many instances, when viewed at an appropriate level of abstraction, amount to a single object seen in differing contexts. Unfortunately the application programmer is not afforded this simple, abstract, view. Instead they must deal explicitly with the many different faces of the same logical object. A programming environment that could provide the programmers with a level of abstraction where such details were only exposed when necessary would again substantially reduce the complexity of the application.

2.1.4 Need to Support Long Transactions

Updating the records for a large company in the ABS-BR can take days of data-entry time, and yet such updates must occur transactionally, so there is a clear need for long transactions to be well supported. In the ABS-BR, application programmers copy data out of the database into a temporary space (called the 'work in progress' or WIP store) for the duration of the long transaction in order to avoid lock conflict problems. At the end of the long transaction updated data is brought back into the database. This is a pragmatic solution to the problem of dealing with long transactions without inbuilt support from the underlying OODBMS. However, this solution does not scale well and by-passes the OODMBS' transactional isolation by taking data outside of the OODBMS' scope. Ideally the OODBMS would provide an abstract solution whereby the distinction between operations within a long transaction and other operations would only be evident from the context—the same query and update code executed in the different contexts would have the appropriate semantics. The programmer should not have to be exposed to long transactions when writing application logic.

2.1.5 Need for Scalability

A system like the ABS-BR places enormous demands on scalability. In large part, the performance demands of the ABS-BR can be characterized in terms of throughput. In principle, this should allow the application programmer to be heavily insulated from the scalability issue, the onus falling on an underlying OODBMS capable of executing user queries at the requisite rate. In fact, the ABS-BR programmers were heavily exposed to the question of scalability as issues like locking strategies and their impact on performance came to the surface. The challenge therefore exists to build OODBMS systems capable of scaling while insulating its users from the mechanisms critical to that performance.

3 Approaching A Solution

Having discussed major challenges raised by the ABS-BR project, we now go on to describe our approach to addressing them. The approach is two-pronged. At the conceptual level, our approach rests on the power of *abstraction* as a tool for conquering complexity, while at a more practical level we depend on the use of *Java* as a powerful environment for constructing a solution.

3.1 Abstraction as a Tool

The enormous complexity of a large object database application like the ABS-BR is a product of the intersection of multiple domains of complexity. While the complexity of the essential system that the ABS-BR seeks to model is moderate, secondary issues that the application programmer must deal with such as *persistence*, *historical versioning*, and *scalability concerns* appear to impact on the system complexity in a multiplicative rather than additive manner. By separating concerns and providing abstractions, the application programmer is able to focus on the primary objective—the core application logic. It is for this reason that we see abstraction as the key to conquering the complexity of such applications. Furthermore, abstraction can be a key to addressing scalability through the clean separation of application and store-level scalability issues. The following sections detail three important applications of this approach.

3.1.1 Orthogonal Persistence

As noted in section 2.1.1, the complexity of the ABS-BR implementation is in large part a consequence of the *impedance mismatch* between the programming environment and the underlying storage technology. This impedance mismatch is manifest in complex mappings between persistent and non-persistent classes and instances. We see this as a prime example of the need for abstraction—abstraction over persistence—and we see orthogonal persistence as a solution to that problem.

Orthogonally persistent systems are distinguished from other persistent systems such as object databases by an orthogonality between data use and data persistence. This orthogonality comes as the product of the application of the following principles of persistence [4]:

Persistence Independence The form of a program is independent of the longevity of the data which it manipulates.

Data Type Orthogonality All data types should be allowed the full range of persistence, irrespective of their type.

Persistence Identification The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system.

These principles impart a transparency of persistence from the perspective of the programming language which obviates the need for programmers to maintain mappings between persistent and transient data. The same code will thus operate over persistent and transient data without distinction. The third principle states that all data required by computation within some system will implicitly persist. This infers a form of transitive persistence and an implicit rather than explicit mode of identifying persistent data. A consequence of this is that orthogonally persistent systems provide automatic storage management through the use of garbage collection with respect to the implicit persistent roots of the system—thus addressing the second key issue raised by the ABS-BR (section 2.1.2).

The abstraction over storage provided by an orthogonally persistent implementation environment would thus facilitate substantially simplified construction for the ABS-BR through the removal of the language/database impedance mismatch and the introduction of automatic storage management. By greatly reducing implementation complexity, the use of orthogonal persistence would also have the significant side effect of reducing maintenance costs.

While orthogonal persistence provides a nice solution to the problem at hand, in fact there exist few implemented orthogonally persistent systems, and, until now, none which are capable of adequately supporting many of the other demands of the ABS-BR such as transaction and version support. The filling of this gap with a portable orthogonally persistent Java (OPJ) [21] has been a major focus of our research group, and is the subject of section 4.1.1.

3.1.2 Orthogonal Versioning

The concept of orthogonal persistence, which has been under development since the early 1980's has a natural analogue in 'orthogonal versioning'. While orthogonal persistence allows the programmer to abstract over the concern of persistence, orthogonal versioning abstracts over versioning. This allows the programmer to write code without regard to the issue of versioning, except insofar as that the programmer finds it desirable to explicitly do so.

In concrete terms, a programmer designing and building a class for the business register is able to focus on the business logic, ignoring the issue of versions. At some high level in the BR implementation, means would be provided for allowing the BR user to define a version context ('*now*', '*last quarter*', '*Q2 2000*', etc.). Operations over the BR would then be executed using the *same code*, from the *same classes*, in the *new context* specified by the user. The implementation code thus remains abstract with respect to versions, except at the few points where it is desirable for the version context to be explicitly set, yet the entire BR operates with respect to fully versioned persistent data.

The first two principles of orthogonal persistence have obvious analogies in orthogonal versioning:

Version Independence The form of a program is independent of the version of the data which it manipulates.

Data Type Orthogonality All mutable data types should be allowed to be versioned, irrespective of their type.[2]

A meaningful implementation of the concept of 'version independence' must provide transparent support for *configuration management*[3] in order to facilitate transparent access to and manipulation of a consistent collection of component versions of a complex object. The importance of configuration management has been widely recognized in areas such as CAD, CASE and web document management systems. One of the consequences of configuration management support is the need to be able to transparently generate new versions of an object. For example, a new version of an object must be created the first time the value of the object is updated within the scope of a new configuration. That is to say that versions are not generated for each update to an object, but rather new versions are generated at the granularity of the instantiation of new configurations.

To the authors' best knowledge the concept of 'orthogonal versioning' is new and previously unimplemented. Our implementation of orthogonal versioning in the context of Java and its application to the BR demonstrator are the subject of sections 4.1.2 and 4.4 respectively.

3.1.3 Abstracting over Distribution and Storage Technology

Ideally, an application developer should not have to be exposed to the particulars of system architecture and storage technology. Whether the technology used to deliver requisite performance is uni-processor, SMP, cluster or some other technology, it should be transparent to the programmer implementing the business logic of the application. Likewise the application programmer should not need to be concerned with the choice of vendor for the underlying

[2] The principle of Data Type Orthogonality applies only to *mutable* types as the concept of versioning makes no sense except where the object may be mutated. Another way of looking at this is that immutable types are trivially versioned, but because the objects are immutable and therefore never updated, all 'versions' of the object present the same value.

[3] The term 'configuration' is used to denote a consistent view of a set of versioned objects. This is analogous to the concept of a 'release' in the context of a software management system—a software release corresponds to a consistent set of files each at a particular (but typically different) point in their own version history.

transactional store. Of course an orthogonally persistent development environment will abstract over the storage technology. However, the *implementer of the orthogonally persistent environment* will have to face this issue, and their approach will impact on the flexibility of the end product. While developing the orthogonally persistent environment reported in this paper, we were faced with these issues of abstraction, and our response to them centered on the use of PSI [6].

PSI was borne out of an appraisal of the problems inhibiting the development of high performance orthogonally persistent systems and the view that a separation of storage and language concerns was an essential step to alleviating those problems. PSI thus represents an abstraction over storage at a low level in the implementation hierarchy. Crucial to the abstraction over storage is the separation of *scalability* issues associated with the store from those associated with the application. This abstraction thus plays a central role in our goal of scalable object server technologies.

A key feature of the PSI architecture is the concept of a 'single image store'. The single image store abstraction allows a storage system to be seen as uniform and single level, when in fact, it may be implemented at different levels, and in a distributed fashion. In the PSI architecture, objects are accessed directly through a cache which is shared between storage and language layers. The PSI interface mediates access to objects in the cache to ensure coherency of data. This mediation is done through a transactional mode of operation, with all data access occurring in the context of some transaction and so being subject to transactional semantics.

The simplicity of the PSI interface makes it possible to efficiently add PSI compliance to transactional object storage systems by adding a thin interfacing layer. A language system that works to the interface can utilize any store that implements the PSI interface, and so has a great deal of portability with respect to storage technology.

3.2 Java as a Development Platform

Java is becoming an increasingly popular choice of programming environment. There are many reasons for this, some of which where behind our decision to choose Java as the basis of our ABS-BR demonstrator implementation project. We view this choice as a significant facet of our approach, and so place it at the same level as our discussion on the role of abstraction in our approach.

3.2.1 Leveraging Java's Attributes

Of Java's many attributes, three stand out as being particularly important in the context of the ABS-BR.

Usability As a modern object-oriented programming language, Java provides its users with numerous features that enhance its usability, including type-safety, encapsulation, automatic memory management, etc. Furthermore, rapid advances in Java virtual machine (JVM) technology allow these features to be delivered to the user with rapidly diminishing performance overheads [2]. In the context of the ABS-BR these usability concerns are significant with respect to the development and maintenance of the ABS-BR application. At present *enormous* resources are devoted to development and maintenance, and issues such as the lack of automatic memory management in C++ are among the greatest concerns.

Portability Another of Java's most outstanding features is portability. Not only may compiled Java code be executed on any platform for which a JVM implementation exists, but micro-applications ('applets') may be transmitted from client to server across the internet, executing on any client platform that has a JVM. This feature is particularly significant in the ABS-BR context, where the execution environment is strongly heterogeneous, ranging from large Unix servers to desktop PCs, and is distributed (through the ABS WAN) across Australia. In such an environment, a server-oriented JVM could be run at the server [2], while clients could simply utilize Java implementations within their browsers. The ABS-BR implementation could be built purely in Java, from the business logic right the way through to the GUI presented to desktop users. This would short-circuit the impedance that currently exists between the Unix-based C++/OODBMS server application and the Windows-based client-side GUI application.

Concurrency Java's support for concurrency is another key feature in the context of the ABS-BR. Concurrency is essential to delivering server-side performance, both in terms of throughput and responsiveness. In this setting, a language such as Java which has strong support for concurrency is a major advantage. The demand for highly scalable JVMs for just such applications has lead to major research efforts within industry to develop high performance server JVMs capable of exploiting hardware and operating system support for concurrency through threads [2], a move which seems destined to deliver such JVMs commercially in the very near future.

3.2.2 Java's Capacity for Extended Semantics

In section 3.1 we discussed the importance of *abstraction* as a tool for resolving complexity, and introduced the concepts of orthogonal persistence and orthogonal versioning as powerful applications of this principle. Both of these comprise *orthogonal semantic extensions* to the programming language. In other words, the semantics of the programming language must be extended to orthogonally (transparently) support additional, semantics (in our case persistence and versioning). The orthogonal persistence literature [4] includes numerous papers addressing the question of how the semantics of persistence can be orthogonally integrated into a programming language.

One of the key breakthroughs in our quest to build an orthogonally persistent Java was the realization that Java has an extremely powerful capacity for semantic extension in its provision for *user-definable class-loaders*. This mechanism allows a custom class loader to be written (in Java), that intercepts (and possibly modifies) classes as they are loaded. This allows the language to be semantically extended through the use of custom class loaders. Significantly, this approach is portable as it does not depend on modifying the compiler or the virtual machine.

4 Engineering A Solution

Having outlined our approach, we can now discuss the technology we developed and applied to our implementation of a BR 'demonstrator'. We begin by discussing our implementations of orthogonal persistence for Java (OPJ) and orthogonal versioning for Java (OVJ). Central to the development of each of these was the semantic extension framework (SEF) for Java, the subject of the next section. Finally we discuss our use of PSI to abstract over storage concerns which allowed us to implement OPJ in a storage platform neutral manner.

4.1 Orthogonal Persistence and Versioning Through Semantic Extensions to Java

In section 3.1 we argued for orthogonal persistence and orthogonal versioning as powerful examples of simplification through abstraction. The realization of each of these in the context of Java depends on *orthogonally extending the semantics of Java*. That is, the normal semantics of Java must be preserved but transparently extended to support new semantics (in our case persistence and versioning). In the following sections we describe the nature of the semantic extensions we have made to Java in order to realize orthogonally persistent Java (OPJ) and orthogonal versioning for Java (OVJ). Section 4.2 explains the technology used to semantically extend Java in an efficient and portable way.

4.1.1 Orthogonally Persistent Java

The primary function of orthogonal persistence is to *abstract over storage*, allowing the semantics of object persistence to become orthogonal to all other object semantics. This leads to the first-order goal for OPJ of making transparent the movement of data between primary and secondary storage, providing the user with the illusion of a single coherent level of storage. Of course there are many second-order goals and requirements for an orthogonally persistent system, but a detailed discussion of these is beyond the scope of this paper. The interested reader is directed to Atkinson and Morrison's review of orthogonal persistence [4], and a detailed account of our OPJ implementation [21].

While our OPJ implementation (ANU-OPJ) is not the first implementation of OPJ [3, 13], a number of important features make it unique: our implementation strategy based on the semantic extension framework gives our implementation portability and access to the fastest JVM technology; ANU-OPJ is fully transactional; ANU-OPJ has a natural and clean approach to identifying persistent objects. These differentiating characteristics come with strong performance results (section 5.1.2). In the remainder of this section we briefly discuss four aspects of the ANU-OPJ implementation before addressing its limitations.

Read and Write Barriers At the core of any orthogonal persistence implementation must lie mechanisms that transparently fetch data from storage on demand and transparently write updates back to storage when necessary. Such mechanisms depend on read and write barriers—traps that transparently catch user attempts to read or update data. Efficient and transparent implementation of read and write barriers are thus essential to any OPJ implementation. The approach taken by ANU-OPJ is in keeping with our theme of transparent semantic extension. Java semantics are transparently extended by the insertion of byte-codes that implement the barriers each time an attempt is made to read an object (e.g. through the `getfield` byte-code) or update an object (`putfield`). These additional byte-codes are added in a stack-neutral manner at class-load time, a modified class loader transforming every `getfield` and `putfield` byte-code accordingly. The efficiency of this approach is largely a function of the virtual machine (VM) technology on top of which it is used—a compiling VM will fold these checks into a small number of machine codes,

while an interpretive VM will be somewhat inefficient. The basic read barrier implementation comprised of a check to see whether the object had already been read. If the object had already been read then nothing was done, otherwise a call was made to the storage layer to read the object in from storage. The basic write barrier consists of the setting of a 'dirty flag' which was checked at transaction commit time. Any 'dirty' objects are written back to the store as part of transaction commit. Implementing read and write barriers for all classes, including system classes was not trivial, nor was it trivial to implement the barriers efficiently. Details of the various approaches used and their performance are included in [21].

Persistence Identification The means of identification of persistent objects is another important facet of an orthogonally persistent system. In most systems, known 'roots' are used as the basis of persistence by reachability—any objects transitively reachable from the 'roots' are implicitly persistent. While other OPJ implementations accomplish this by introducing *explicit* user-definable and retrievable 'named roots' [3, 13] , ANU-OPJ takes an alternative path, making non-transient class static members *implicit* roots. Any object transitively reachable from the static class member of a class is implicitly made persistent. This approach seems to be truer to the *principle of persistence independence* (section 3.1.1) as it allows any Java code to become transparently, implicitly, persistent without any modification whatsoever. The value of this feature is borne out in our experience with the ABS-BR demonstrator (section 4.4) which was built from scratch without explicit concern for persistence in a conventional Java development environment, and yet ran persistently, as intended, without modification once the non-persistent Java was replaced by ANU-OPJ.

Concurrency and Transactions Concurrency is of utmost importance to our motivating example, the ABS-BR, and yet the intermix of concurrency and persistence has been a major stumbling block for orthogonal persistence [7]. The root of the problem lies in the conflicting approaches to concurrency traditionally taken in the programming language and database fields. On one hand programming languages typically adopt, for reasons of flexibility and expressibility, a *cooperative* concurrency model centered on the use of shared variables. On the other hand, for reasons associated with the need for coherent stability, database systems typically adopt a concurrency model centered on the use of *isolation* through transactions. The 'chain and spawn' transaction model presented in [7] provides a way out of a difficult problem[4] that arises when trying to make an orthogonally persistent environment fully transactional and capable of inter (and intra) transactional concurrency. This model is provided by ANU-OPJ, allowing users to work in a concurrent and yet fully transactional environment. Transactional isolation is cheaply ensured by ANU-OPJ by leveraging Java's existing class-loader isolation. ANU-OPJ ensures that each new transaction is run in the context of a different class loader and Java ensures that Java objects remain isolated on a transactional basis. Transactional guarantees provided by the underlying store are also leveraged by ANU-OPJ to ensure that correct transactional semantics are observed.

Implementation Strategy The key to the ANU-OPJ implementation strategy lies in its use of the semantic extension framework, SEF (section 4.2). The semantic extensions embodied by ANU-OPJ are injected into classes at class loading time, thus giving normal Java programs orthogonal persistence semantics. This contrasts strongly with the dominant approach of modifying the Java virtual machine, which is motivated by the performance gains to be made by building the semantic extensions into the virtual machine at the lowest level. Interestingly, ANU-OPJ is able to outperform an OPJ built using this approach (section 5.1.2). This appears to be the product of two factors. First, in a compiling JVM, such as one with a just in time compiler (JIT), the byte-codes added by the SEF are compiled and optimized just as they are in the 'built-in' approach. Second, for the 'built-in' approach to maintain its advantage, the semantic extensions must be 'built-in' to each new release of the parent JVM—a demanding task given that the semantic extensions must be added to the component of the virtual machine most targeted for improvement (the interpreter and/or compiler).

Limitations ANU-OPJ is relatively young, and so holds plenty of scope for improvement. Aside from performance improvements, ANU-OPJ in its current form also embodies a number of limitations. While most of these are minor, its current incapacity to make threads persistent is at odds with its claim to orthogonal persistence[5]. However, we believe that this problem can be relatively easily overcome by borrowing an approach used in the context of thread migration [15].

[4] The problem arises because when all computation is transactional and inter transactional concurrency is desired, some form of nesting is required. Yet, the basic building block of transactions—the ACID transaction [14]—cannot be nested because of a contradiction between the atomicity of the parent and the irrevocability of the child. For further information, see [7].

[5] This limitation is common to the two other major OPJ efforts [3, 13].

4.1.2 Orthogonal Versioning in Java

Object versioning is a natural concept in many object-oriented applications. Versioning becomes important in applications where an exploratory or iterative approach over the object store state is needed. In the ABS-BR demonstrator, we utilize object versioning both as a means of facilitating queries and updates against historical data, and as a practical means of efficiently implementing long transactions. This is all done through our implementation of orthogonal versioning for Java (OVJ), the subject of the remainder of this section.

As with OPJ, OVJ consists of semantic extensions to Java, and like OPJ, OVJ utilizes the semantic extension framework, SEF (section 4.2) to realize these semantic extensions. Here we describe OVJ as implemented on top of ANU-OPJ (section 4.1.1 above), although OVJ can be integrated into other modes of persistence support for Java such as an ODMG-3 transparent persistence implementation (section 5.2). Our OVJ does not depend on explicit support for versioning within the underlying object store. In fact, our first implementation of OVJ was built on top of the SHORE OODBMS [11], which has no support for versions.

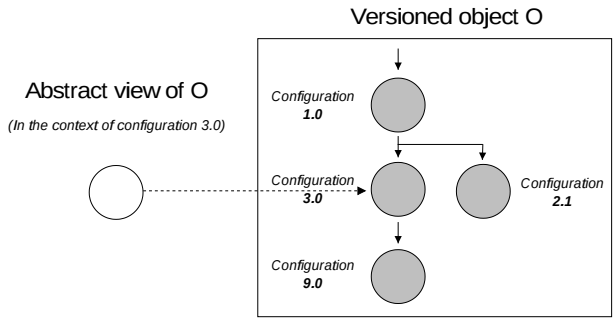

Figure 2: A versioned object in OVJ. The user's perspective (left) abstracts over the complex version hierarchy (right). The particular version of the object seen by the user is an implicit function of context (in this case configuration 3.0).

Figure 2 illustrates the abstraction provided by OVJ, and depicts the relationship between versions and 'configurations' (section 3.1.2). In this example, the global environment includes many different configurations, while the depicted object only has versions in four of these. The power of the abstraction over versions provided by OVJ is seen in the left side of the digram, which presents a simple view of the object in a particular configuration *context*, while the underlying situation (which OVJ abstracts over) is the complex version tree depicted at the right. The importance of a configuration context is that it allows the denotation of versions to be implicit rather than explicit, which is essential to our goal of transparency. Having established a configuration context (perhaps implicitly), user computation proceeds, oblivious to the issue of versions, over a normal object graph which in fact corresponds to some projection of a versioned object graph into the Java heap space. In the remainder of this section we discuss four major aspects of our OVJ implementation.

Runtime overhead An important goal in designing OVJ was minimizing the performance impact on objects that are naturally transient and so are unaffected by versioning. This is achieved by only applying versioning to objects at the time that they first become persistent. Any object that because of its transience never becomes persistent, avoids most of the overhead associated with versioning. However, some overhead is unavoidable, even for transient objects. Under OVJ all instances[6] include an additional field that is used for storing a reference to version information for that object, regardless of whether the particular instance is sufficiently long lived to be subject to versioning. Additionally, the read and write barriers of OPJ are extended slightly (all objects are subject to the barrier overhead). In addition to these overheads which are imposed on all instances, those instances that are sufficiently long-lived to be versioned are subject to a small space overhead that is used to link instances that correspond to different versions of the same object. A runtime overhead must also be paid on the first access to an instance that corresponds to a specific version of an object.

Orthogonality With Respect to Versioning and Persistence In order to ensure the persistence of all versioned objects, in the context of OVJ, OPJ's approach to defining persistent roots is extended by making all *versions* of

[6] Although under OVJ *all* classes are subject to versioning in order to maintain orthogonality, the underlying versioning framework, which we describe here, allows versioning to be scoped with respect to some part or parts of the Java class hierarchy.

non-transient static class members roots (in OPJ persistence is defined in terms of reachability from the static class members). In order to meaningfully fulfill the principle of version independence, OVJ must provide transparent support for configuration management. In other words, there must be means for transparently accessing and manipulating a consistent collection of component versions of a complex object. This approach provides a transparent and simple abstraction over versioning by allowing the basic versioning classes to be extended and customized. The versioning framework on which OVJ is constructed also provides a very general and powerful approach to versioning that has application to many different contexts, and that need not be as transparent.

The Configuration Hierarchy A configuration represents a set of versioned object versions, each configuration containing no more than one version of a given object. Each configuration except the first has a parent, and each may have any number of children. Thus, a linked set of configurations corresponds to a n-ary tree, and a linear sequence of configurations corresponds to a branch. In figure 2 a subset of the nodes comprising a configuration tree can be seen in the version tree of object O. While the very purpose of OVJ is to provide a means of abstracting over versions, it is necessary to provide the user with some way of explicitly controlling the configuration hierarchy. Minimally, such control might just involve a means for instantiating new configurations (akin to ABS-BR historical snapshots). OVJ also provides a much more powerful facility which allows the user to generate and manipulate a configuration hierarchy in quite general ways, including branch generation and configuration tree navigation. The current implementation does not support automatic configuration merging (which derives a new configuration by merging two or more existing configurations). However, a special class of configuration exists, whose instances may have more than one parent configuration (configuration merges create cross-links in the configuration n-ary tree). OVJ also supports the concept of configuration 'freezing'. A configuration may be labeled as frozen, with any attempt to write to a member of that configuration triggering an exception.

OVJ's Underlying Object Model In this section we describe something of the model underlying OVJ's implementation. While this detail is not exposed to the user of OVJ, it may serve to give the reader a better understanding of some of the issues faced in implementing OVJ. The basis for OVJ is a versioning framework which is implemented by semantically extending Java classes at class loading time through use of the semantic extension framework. For a class to be versioned, it must be (transparently, by the SEF) either made to inherit (directly or indirectly) from the class `VersionView` (or made to implement the interface `ObjectVersioning`). Each instance of `VersionView` includes a reference to a configuration instance and to the specific version instance associated with the given configuration[7]. (Such an instance corresponds to the object at the left of figure 2, which in practice would also refer to the configuration instance corresponding to 3.0.) Each of the specific version instances are instances of an automatically generated class corresponding to the base class of the object, but containing only those fields that must be made persistent and with the addition of fields that link the various versions of the instance (these instances correspond to the gray objects at the right of figure 2). This approach of dynamically creating new classes in a manner that is transparent to the user is common to OPJ, and although beyond the scope of this paper, is described in some detail in [21].

Versioning API OVJ provides a versioning API for all objects, which includes the following methods:

```
public ObjectVersioning getClosestVersion(String name);
public ObjectVersioning getCreateVersion(String name);
public void deleteCurrentVersion();
public ConfigurationInterface getConfiguration();
```

The first two, `getClosestVersion()` and `getCreateVersion()`, both return an instance of the object corresponding to the given configuration. (If the configuration does not exist an exception is thrown.) If such an instance version does not exist the first method will return the closest ancestor and the second will create a new version. The third method separates the associated specific version from the other versions of the object. The last returns the configuration associated with the instance it is called over.

4.2 SEF: The Semantic Extension Framework

Fundamentally, both OPJ and OVJ constitute *semantic extensions* to Java, in one case to support persistence, in the other to support versioning. We implement these extensions through use of the Semantic Extension Framework (SEF).

[7] An instance of `VersionView` may also reference a *virtual configuration.* A virtual configuration represents the set of versioned object versions that satisfy a given condition. Virtual configurations typically span a configuration sub-branch.

The SEF is a *general* framework that was developed after our experience with ad-hoc approaches which we used in our initial implementations of OPJ and OVJ. Aside from its generality, the SEF is distinguished by its use of Java's class loading mechanism, which allows the extensions to be applied dynamically and gives the process portability. Despite its generality, simplicity and portability, we show in section 5.1.2 that it can outperform a solution based on direct modification of the Java virtual machine.

4.2.1 Semantic Extension: Goals and Implementation Constraints

The 'write once, run anywhere' philosophy has been central to the success of Java. The approach we have taken to semantic extensions is in keeping with this philosophy of portability. While this could be viewed as a limitation (restricting the programmer to the mechanisms offered by standard Java rather than extending Java) it enables implementations to exploit the best Java technology available (as it can run on any virtual machine and use any compiler).

A system that extends the semantics of Java should maintain fundamental properties of the language and runtime system including: *separate compilation*, *dynamic class linking*, *modularity*, and *portability*. Original language semantics must be maintained in order to preserve the properties of separate compilation and dynamic class linking. New properties, such as fields and methods, may be added to a class only if the existing API contracts are respected. Furthermore, any semantic extensions should compliment existing semantic change mechanisms, i.e. inheritance and overloading.

An important objective is that the semantic extensions can be *dynamically* composed. Many semantic extensions such as program instrumentation and profiling are volatile by nature. These semantic extensions should be applicable to other more 'permanent' semantic extensions, such as orthogonal persistence or object instance versioning. Consequently, the specific set of semantic extensions to be applied may only be known at runtime, emphasizing the need for dynamic composition of semantic extensions. In fact, it could even be of interest to dynamically extend the semantics of classes already loaded (e.g. the dynamic instrumentation of a program already running). However, the JVM specification forbids modifications of classes after they have been loaded, so the only possible solution in this case is the use of a modified JVM with advanced support for introspection.

4.2.2 Semantic Extension Through Byte-code Modification at Class-Load Time

Until now, efforts to extend Java's semantics to support persistence could be classified in terms of two approaches: those that modify the Java virtual machine [3, 13, 17], and those that use a modified compiler or post-processing to produce modified byte-codes [16] (in some cases both approaches have been employed). The first approach clearly violates our the goal of portability—the semantics will only be available on modified virtual machines. The second approach produces portable byte-codes, but requires each producer of semantically extended code to have access to a modified compiler or preprocessor. Moreover, the compilation approach precludes the dynamic composition of semantic extensions. Byte-codes produced under the second approach indelibly embed the semantic extension, and so are no longer able to exhibit their original semantics.

The approach we have taken is to transform byte-codes at class load time, a technique that has been used in other contexts [8, 1], but until now has not been applied in the context of persistence. The approach has numerous advantages including portability with respect to the target machine and portability of the users' byte-codes. (Unlike the second approach listed above, the byte-code modifications happen inside the JVM and so are not visible to the user and do not affect the source byte-codes). The basis for the technique is Java's support for the use of user-defined class loaders which may modify the content of a class file before loading the class into the JVM. The virtual machine will apply all of the usual checks to the modified class. By using a user-defined class loader to introduce semantic change, standard compilers and virtual machines may still be used.

4.2.3 A Declarative Semantic Extension Language

As we have said already, our initial implementations of OPJ and OVJ were made by constructing custom class loaders that made *ad-hoc* byte-code transformations. This experience with an ad-hoc approach directly motivated the development of the SEF, which has been used as the basis for our subsequent OPJ and OVJ implementations. Our distaste for the ad-hoc approach was driven by many factors, but the fundamental one was that byte-code transformations are difficult and error prone. A simple mistake during the transformation process can destroy type safety or the semantics of the program, and may lead to the byte-code modified class being rejected at class loading time. So just as a compiler raises the level at which the programmer is able to specify a program's semantics (through the use of a high level language), the SEF raises the level at which semantic extensions may be specified. In each case the programmer is able to avoid the complexity and hazards of working at too low a level.

The declarative language of the SEF is characterized by a class-based naming protocol for defining the *scope* of semantic extensions and the use of Java to specify the *semantics* of the extensions. Thus users encode semantic extensions in Java classes, and the naming of the classes reflects the scope of the extension. For example, the semantic extensions specified in a class named `java$lang$Object$` will be applied to all classes that inherit from the Java class `java.lang.Object`.

Semantic extensions can include the addition of new fields and/or new methods to the modified class(es). The SEF has a convention for adding specially named methods that function as 'triggers'. Trigger methods are automatically invoked on the occurrence of certain events such as field accesses, array accesses, method invocations, and parameter loading. Both 'pre' and 'post' triggers are supported, allowing the trigger method to be executed either immediately before or immediately after execution of the triggering event. Trigger methods are defined through the use of a simple naming convention—`pre$getfield$()` identifies a method which will be invoked prior to any execution of a `getfield` byte-code in classes falling within the scope of the extension definition (similarly `post$putfield$()` will be invoked after any execution of a `putfield`).

```
public class java$lang$Object$ {
    protected final java.lang.Object pre$getfield$() {
      if(Tracer.showAccess(this))
        System.out.println("Going to access object "+this);
      return this;
    }
}
```

Figure 3: A declaration of a parametric semantic extension to achieve trivial instrumentation.

These concepts are illustrated in figure 3. The name of the class (`java$lang$Object$`) indicates that the *scope* of the semantic extension is the class `java.lang.Object` and all of its sub-classes. The *semantics* of the extension are limited to a single method `pre$getfield$`, which, as a special trigger method, will be invoked immediately prior to the occurrence of any `getfield` byte-code in any class within the scope of the extension. In this example, the trigger instruments a trace message, with the `showAccess` method of the class `Tracer` dynamically determining whether or not the message will appear.

The semantic extension framework is invoked each time a user class is loaded. This action triggers a special semantic extension class loader to search for and load any semantic extension classes that are applicable to the user class being loaded (according to the scoping rules already described). The semantic extension class loader relies on the same visibility rules as all other Java class loaders. This means that a whole suite of semantic extensions may be 'turned on' or 'turned off' by simply including or excluding the path containing extension classes from the class paths visible to the JVM (which can be done trivially either by changing the `CLASSPATH` environment variable or by setting the `-classpath` option on the JVM command line).

4.2.4 Further Implementation Issues

A detailed account of the SEF and its implementation is beyond the scope of this paper—the interested reader is referred to [21]. However, we will briefly mention two implementation issues. The first of these is the way that the SEF deals with system classes. Because Java system classes are not directly modifiable, the SEF uses a proxying approach whereby references to the target system class are transparently redirected to a proxy for the system class. Although the SEF cannot *modify* any of the system class methods, the proxy mechanism can be used to *redefine* any of the methods, calling back to the unmodified system class if desired. Extensions defined with respect to a system class are, of course, applied to any class that inherits from the system class (as seen in the example in section 4.2.3, where the `pre$getfield$` trigger will be propagated to all user classes). The second important issue is the impact of the SEF on Java's introspection mechanisms. Because the SEF rewrites classes and in many cases generates new classes, methods such as `getClass()` may return unexpected results. Fortunately such confusing side-effects can be automatically rectified by the SEF by semantically extending Java's introspection mechanisms to reflect the desired behavior.

4.3 PSI: Abstracting Over Storage

By abstracting over storage, the application developer is freed from explicit concern for the characteristics of the underlying storage architecture—whether the system is scalable, distributed, what the interface of the particular system

is, etc. Furthermore, as the demands of the application change the choice of most suitable storage platform may also change. If the application, through a layer of abstraction, has become independent of the particular storage platform, the storage system may be changed without perturbing application code. In this section we identify PSI, which is an instantiation of the *transactional object cache* architecture.

Central to the architecture is the *cache*, to which the underlying store, the language run-time and the application may have direct access. A *transactional interface* mediates the movement of data in and out of the cache, giving the language run-time (and through it the application) transactional guarantees of atomicity, isolation, coherency and durability with respect to its accesses to the cache. A layering of storage and programming language concerns is explicit in the architecture.

The explicit acknowledgement of the centrality of the cache and the provision for mediated direct access to it contrasts with other architectures for persistent systems [22, 23] which provide the abstraction of a 'persistent heap', implemented with a relatively expensive procedure call interface for data access. This aspect of the transactional object cache architecture makes it far more conducive to high performance implementations than such alternatives.

By providing an abstraction over the object store through layering, the architecture is transparent to the distribution of the underlying store, and coupled with the concurrency control delivered by the transactional interface, facilitates

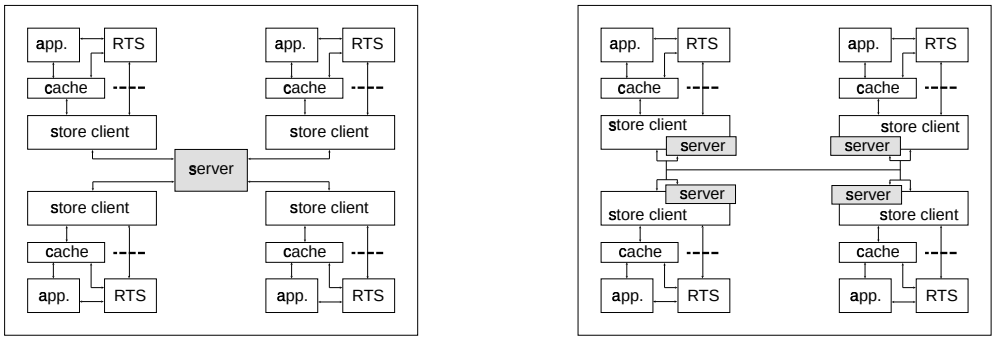

Figure 4: Client server (left) and client peer (right) realizations of the underlying object store. The abstraction over store architecture provided by the transactional object cache architecture gives distribution transparency to the client application and run time system. PSI is an instantiation of the RTS/store interface visible at each node.

the implementation of multi-user client-server or highly scalable multi-processor implementations independently of the language run time/application [6]. We characterize this property of distribution transparency from the perspective of the run time and application with the term *single image store*.

4.3.1 PSI: A Transactional Object Cache Interface

At the heart of the transactional object cache architecture is an interface between the language and store layers of a persistent system. A realization of the architecture is thus dependent on the identification of a suitable interface. To this end we have developed PSI, a software interface definition [6].

In designing PSI, we sought to balance a number of objectives: to flexibly support the needs of persistent programming languages (PPLs); to strongly separate storage and programming language concerns; and to admit small and fast PPL implementations. This combination of objectives presented a constrained trade-off space for the interface design, within which PSI represents just one point.

The underpinning for the PSI definition is an abstraction of the transactional object cache architecture [6]. The abstraction is based on three distinct facets of the behavior of the transactional object cache: *stability*, *visibility*, and *availability*. A number of abstract operations over the store such as ReadIntention, WriteIntention, and Commit are identified and their semantics defined in terms of the three abovementioned domains of behavior. The well-defined semantics of these primitives were then used as the basis of a definition of the semantics for each of the PSI calls.

Having defined the PSI interfaces, we have constructed bindings to the SHORE OODBMS [11] and to Oracle's object cache and relational interfaces. In addition, a high performance object store under development within our lab also has a PSI interface. Thus the scalability of the ABS-BR demonstrator is in large part a function of the choice of underlying storage system.

4.4 Implementing the ABS-BR Demonstrator

Recall that our goal in the ABS-BR project was to expose ourselves to a large 'real-world' problem and then to develop and test new technologies in the context of that problem. The ABS-BR and the challenges that it gives rise to are the subject of section 2, and in section 3 we discussed an approach based on the use of abstraction as a methodology and Java as an implementation context. In the preceding portion of this section we have outlined the development of technologies which put our approach into practice. We can now discuss the implementation of a first prototype of the ABS-BR demonstrator, which is a realization of key components of the ABS-BR using the abovementioned technologies.

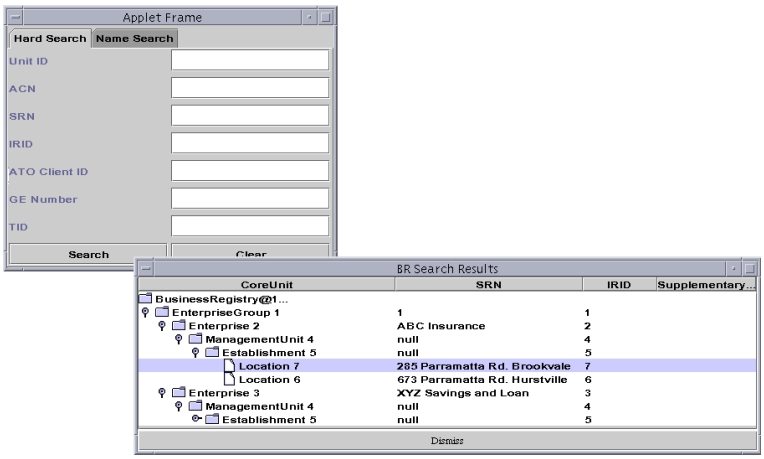

Figure 5: User interface of the OPJ/OVJ-based ABS-BR demonstrator. Shown are the startup window and a query window displaying a set of company hierarchies that satisfy a user query.

The demonstrator is built in Java and uses OPJ and OVJ for transparent support for versioning and persistence and, through OPJ, uses PSI to gain portability with respect to, and leverage the scalability of, the storage back-end. The application has a user interface built using Swing (figure 5), and allows the user to: navigate the BR's core business unit hierarchy; create, query and update company information; use both long and short transactions; and access historical snapshots of the database.

The demonstrator was constructed over a period of four months by a single programmer. The system is written entirely in Java and was developed using a popular integrated development environment. Because of the orthogonality of OPJ and OVJ, the code is pure Java and is devoid of the explicit persistence and version management code which are dominant contributors to the complexity of the original ABS-BR. In the remainder of this section we discuss three significant aspects of the demonstrator implementation.

4.4.1 Implementation of Long Transactions

Long transactions are very important to the ABS-BR because updates to large companies can take days of data-entry time. As discussed in section 2.1.4, in the absence of support from the underlying OODBMS, this need is addressed in the ABS-BR by copying data out of the database into an external space called the work in progress (WIP) store for the duration of the updates. This pragmatic approach is far from ideal because it does not scale well (all data used must be copied), and it bypasses the database's transactional mechanisms.

Our solution is to view long transactions as an application of object versioning. Looked at this way, a long transactions corresponds to a branch in a configuration hierarchy and the commit of a long transaction corresponds to a configuration merge (bringing the branch back into the main-line). Because a branch in a configuration hierarchy is a *logical* duplication of the database, (short) transactions operating over the mainline and branch will not conflict and so can operate concurrently. Yet because the duplication is *logical* rather than *physical* no unnecessary copying occurs. Operations within the long transaction are composed of normal transactions with respect to the appropriate configuration branch of the object database. Using this approach, a user query can occur concurrently to a long-running update to one of the companies queried without conflict. While in general branch merges can generate conflict which must be manually resolved, conflict resolution is avoided in the ABS-BR prototype by using a simple system of high-level locks to ensure that (rare) conflict-generating transactions do not execute concurrently.

Because this solution is based on the use of versioning through OVJ, aside from calls to start and end a long transaction, it is completely orthogonal to user code.

4.4.2 Implementation of Historic Versioning and Dependent Source Feedback

The need for historical versioning is a key requirement of the ABS-BR (section 2.1.3 and figure 1). In particular, the ABS-BR has the concept of 'historic snapshots' (globally consistent historical views of the database), and 'common frames' (means for retrospectively updating historical data). Both historical snapshots and common frames were trivially implemented through the application of OVJ's configuration management features. Branched configurations are a natural solution for the implementation of historic snapshots. Any configuration in the initial branch (main-line) is a historic snapshot, except for the last one. A configuration status is frozen once a descendent configuration in the same branch is created. A common frame corresponds to a new branch created to allow changes to a historic snapshot.

OVJ's configuration management also provides a solution to the problem of dependent source feedback (section 2.1.3). 'Virtual configurations' allow configurations to be defined conditionally, and so in the case of dependent source feedback (DSF), virtual configurations (section 4.1.2) can be used to exclude (or include) object versions which are affected by DSF from any query[8].

5 Analysis

The goal of the ABS-BR demonstrator was to provide a context for evaluating the technologies we have developed and the approaches that they embody. This section focuses on that evaluation and concludes with a brief discussion of alternative approaches and directions for future work.

5.1 Evaluation of the ABS-BR Demonstrator, OPJ and OVJ

An evaluation of the ABS-BR demonstrator must include both qualitative and quantitative aspects of both the demonstrator development and the underlying technologies.

5.1.1 Productivity

In large part ABS-BR productivity concerns stemmed from a high degree of code complexity induced by persistence, versioning, and explicit memory management. We briefly analyze the ABS-BR and the demonstrator with respect to two aspects of overall productivity—development and maintenance costs.

Development The ABS-BR was developed using state of the art commercial OODBMS technology. The development, including a detailed analysis phase, took a team of around fifteen programmers three years (45 person years). The business logic of the application comes to more than 90,000 lines of C++ (not including the graphical user interface). Much of this is accounted for by the use of templates for the generation of an object versioning mechanism, maintenance of explicit persistent to transient mappings and explicit memory management through `alloc`/`free`.

By contrast, the development of the ABS-BR demonstrator prototype took one programmer four months. The demonstrator implements the core ABS-BR schema and about 25% of the total functionality in just 3,400 lines of Java. An additional 3,050 lines of Java implement an advanced graphical user interface using Swing (the GUI includes support for drag and drop, hierarchical displays etc.). The ease of development was highlighted by the fact that the first version of the demonstrator was developed as a non-persistent Java application entirely independently of OPJ and yet ran persistently *without modification* once OPJ became available[9]. Although the demonstrator does not implement user help, data migration and other such peripheral functionality, the simplicity of the demonstrator implementation indicates that the use of Java, OPJ and OVJ facilitates a *substantially* simpler (and thus cheaper) development process.

Maintenance While it is not possible to make meaningful comparisons about the *actual* maintenance costs of the respective systems, two characteristics stand out as indicators that maintenance costs for the demonstrator would likely be substantially lower than for the ABS-BR. First, the complexity of the code is dramatically reduced, yielding substantially shorter and simpler code. For example, the class `coreu` which implements the important CoreUnit in the BR application, is 3113 lines of C++ code in the ABS-BR and 1213 lines of Java in the demonstrator. Since code size

[8] At the time of writing, support for DSF-free queries through virtual configurations is still under development.

[9] OPJ was being developed concurrently and so was not available until after the first demonstrator prototype was ready.

and code complexity are key factors in software maintenance costs, it seems likely that the demonstrator code would have appreciably lower maintenance costs. Second, the implementation of versioning at the level of application code through templates adds a great deal to the complexity of the ABS-BR code and, again, is likely to be a significant ABS-BR maintenance cost that will not be incurred by the demonstrator.

5.1.2 Performance

Ideally we would present results directly comparing the performance of the ABS-BR with that of a complete ABS-BR demonstrator implementation. However, while the demonstrator currently implements the most important functionality for such a comparison, it is not possible to access ABS-BR source data, and for a variety of reasons it would not be practical to benchmark the ABS-BR against some other data source. Instead we evaluate the performance of the key underlying system—OPJ. The primary role of OPJ is as a persistent object system, so we compare it here against a number of alternative such systems using a standard OODBMS benchmark, OO7 [9].

In the results that follow, we include performance figures for our first two implementations of OPJ (ANU-OPJ-shell and ANU-OPJ-facade), each of which used different swizzling policies [21]. The results show performance problems for both system when executing 'cold'[10]. In the case of ANU-OPJ-shell, we attribute this largely to eager memory consumption for the allocation of shell objects in the swizzling process. Whereas for ANU-OPJ-facade, we believe that the poor results are largely due to extensive use of the Java Native Interface (JNI) and the substantial cost incurred on each transition of the Java/C++ interface. Additionally, some of the transformations used in the façade approach may reduce the opportunities for code inlining and optimization by the just in time compiler (JIT) [21].

In both of these implementations native methods were used for the time consuming tasks associated with read barriers, such as object faulting and reification. However, instrumentation of the system demonstrated that the barriers performed substantially better when the kernel of the barrier was implemented in Java rather than as native methods. This is attributed to the substantial overhead of native method invocation and the capacity for the JIT to optimize frequently executed code. In future implementations we hope to substantially reduce our use of JNI and C++.

Benchmarking Environment We have used the OO7 benchmark [9] to evaluate the systems. The results presented here compare the performance of ANU-OPJ (ANU-OPJ-shell and ANU-OPJ-facade), PJama [3] (versions 0.5.7.10 and 1.2), PSI-SSM (a SHORE-based implementation using C++), and Java running without persistence. The PJama implementations are unable to take advantage of JIT technology, whereas ANU-OPJ can leverage this technology to produce competitive performance and portability. The 'small' OO7 benchmarks are used as we were not able to run the 'medium' database over any version of PJama or PSI-SSM.

It is possible that the hot [11] times could degrade as the database size is increased. In this case, techniques such as object cache eviction and promotion will become a necessity. None of our implementations support these techniques, but this support could be easily built on top of the normal Java garbage collection (using finalizer methods and weak references), with the support of an object cache that can flush objects and still maintain their locks.

The same Java code was used for the ANU-OPJ, PJama and JDK systems with only minor modifications required for each. For the base code to run on either ANU-OPJ-shell or ANU-OPJ-façade it was only necessary to insert a call to `Chain()` at the point where the benchmark required a commit. The non-persistent version (JDK 1.2.2) needed minor changes to allow the 'generation' and 'benchmark' phases of OO7 to occur in a single execution. For PJama it was necessary to add code which opened the store, retrieved the persistent roots, and called `stabilizeAll()` at the point where the benchmark required a commit.

For PSI-SSM, the OO7 benchmark was implemented in C++. The implementation does not use swizzling, but instead explicitly translates persistent references to pointers at each traversal, and makes explicit update notifications. 'Smart pointers' [20] were used to perform the reference translations, affording a degree of syntactic transparency.

The benchmarks were executed on a single Sun Ultra-170 with 128MB of RAM and separate hard disks for the persistent store and log. Both version 0.5.7.10 (which required JDK 1.1.7) and version 1.2 of PJama were used. JDK 1.2.2 (with the Hot Spot JIT) was used to execute the ANU-OPJ-shell, ANU-OPJ-façade and non-persistent (JDK) versions of the OO7 benchmarks. ANU-OPJ-shell, ANU-OPJ-façade and PSI-SSM used the Shore storage manager.

Performance Results Each impulse reported corresponds to the normalized average execution time for ten executions of a particular benchmark. The benchmarks reported here are traversals t1, t2c, t3b, t4, t5do, t5undo, t6 to t10

[10] Cold execution times are for the initial run where the data has not been faulted into memory and must be retrieved from the underlying store.

[11] Hot execution times are for runs where the data has already been faulted into memory.

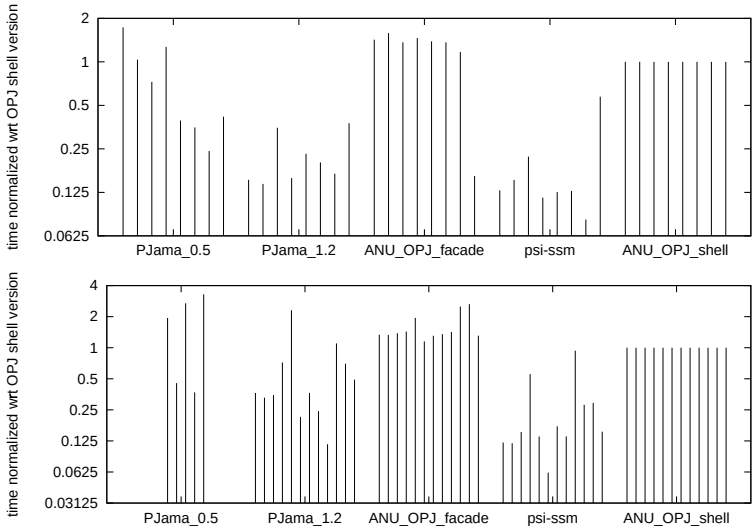

Figure 6: Cold query (top) and traversal (bottom) times relative to corresponding ANU-OPJ-shell time.

and wu and queries q1 to q8, in this order[12]. The results are presented separately for the traversals and queries on account of markedly different characteristics of the two benchmark groups.

The cold execution results in figure 6 indicate that ANU-OPJ implementations performs worse than PJama1.2 when cold (2.2 times slower than PJama version 1.2 on average). We attribute these results to the excessive context switching between Java and C++ using JNI.

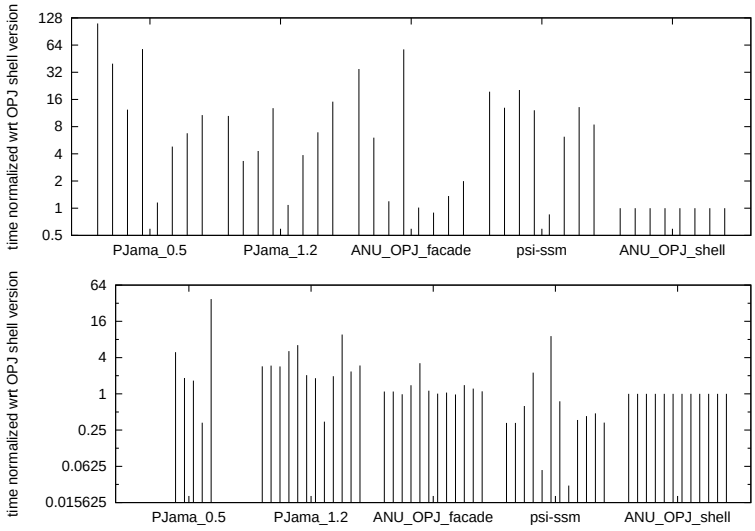

Figure 7: Hot query (top) and traversal (bottom) times relative to corresponding ANU-OPJ-shell times.

The hot execution results in figure 7 indicate that ANU-OPJ-shell implementation performs well when hot (5 times better than PJama version 1.2 and 3 times better than the ANU-OPJ-façade implementation on average), outperforming both PJama implementations in almost all the operations. The ANU-OPJ-shell performs better than any other implementation in read only operations (almost 8 times better than PJama 1.2 on average), even outperforming a C++ based implementation over the same store. We attribute the strength of these results to the JDK1.2.2 JIT compiler and to the cost of explicit reference translation in the C++ implementation.

[12]It was not possible to present results for the complete suite of OO7 benchmarks. Benchmarks t2a, t2b and t3a did not run in either ANU-OPJ implementation nor PJama. Benchmarks ‘i’ and ‘d’ did not run due to an unidentified failure in the Shore storage manager.

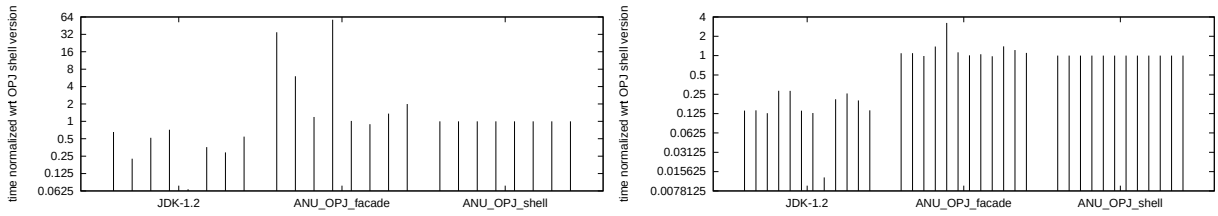

Figure 8: Hot query (left) and traversal (right) times for a non-persistent implementation (JDK) and ANU-OPJ-façade relative to corresponding ANU-OPJ-shell times.

Read and Write barrier overhead Figure 8 compares the ANU-OPJ shell and façade implementations with a non-persistent implementation of the OO7 benchmark operations. This graphic demonstrates a moderate overhead for all query operations (150 percent on average). However, traversal operations were much more expensive (370 percent on average), largely as a consequence of greater transaction commit costs for that group of benchmarks. We believe that the query results are significant, because they suggest that the runtime efficiency of the Java language environment is unlikely to be a stumbling block to the use of Java in the context of persistent data management.

5.1.3 Scalability

We have not been able to make a detailed scalability analysis of the ABS-BR, OPJ or OVJ, although we have shown that the demonstrator will scale to at least half a million objects.

However, the key scalability properties of the ABS-BR lie in its *architecture*, and in large part are independent of OPJ or OVJ but depend instead on the scalability of the underlying object database. The use of the PSI interface allows OPJ to utilize any underlying store that supports the interface—we have trivially constructed bindings for a number of databases, including Oracle. The fact that user code is unaffected by the choice of OODBMS liberates the developer, enabling the user to choose the OODBMS that best suits their demands at any time without impacting *any* of their application code (of course a once-off data migration will probably be necessary).

5.2 Commercial Alternatives and Related Work

The focus of this paper has been the challenges of high performance object database applications and the development of technologies that address them. We have presented some quite specific solutions to these problems and demonstrated their utility. Although our solutions have direct commercial application, at this stage they lie in the realm of academic research. However, there are alternative approaches available, including commercial ones, some of which were independently developed during the course of our work. In this section we briefly look at some of these.

5.2.1 OPJ Alternatives

To this point there have been no commercially available persistent object systems that fully implement orthogonal persistence for Java. Some would argue that this reflects a long-held commercial perspective that orthogonal persistence is an impractical technology [10], and it certainly seems true that orthogonal persistence researchers have not been sufficiently pragmatic in their efforts to have the technology commercialized. Nonetheless, recent developments indicate that technological advances (particularly with respect to Java) and growing demand for object technology are bringing vendors ever closer to the orthogonal persistence ideal. This shift is most notable in a progression of ODMG standards, the most recent of which, ODMG 3.0, has led to the proposal of a new standard for persistence of objects in Java—the Java Data Objects standard [27][13]. This specification is expected to play a fundamental role in the portable implementation of container-managed persistence in Enterprise JavaBeans (EJB) [26] servers. Thus Java Data Objects implementations probably correspond to the closest commercial offering to OPJ, and could serve as a close substitute to OPJ in developing a solution such as the one presented here. Indeed, we have used the SEF to develop our own JDO implementation and have built a version of the ABS-BR using JDO rather than OPJ.

Unfortunately JDO has a number of serious deficiencies. Notably, although JDO implements persistence by reachability, persistence is *not* type-orthogonal. Thus an object may be reachable from a persistent root but of a non-persistent type, leading to dangling pointers in the persistent object graph. Furthermore, JDO does not have the pure

[13] At the time of writing the JDO proposal had just been 'approved for development' under the Java Community Process (JSR-12). See `http://java.sun.com/aboutJava/communityprocess/jsr`

transactional model of our OPJ implementation, but rather intermixes transactional and non-transactional computation. This has two significant consequences: transactional isolation is *not enforced* within the JVM, and in order to implement abort semantics in a manner that is visible to non-transactional computation, JDO implementations must explicitly restore cached objects which were updated in any aborted transaction.

5.2.2 OVJ Alternatives

Some OODBMS, such as Ode [19], SOP [18] and ONTOS [24], provide object versioning support as part of their programming language environment. Most of these products do this through hard-coded version semantics and lack support for configuration management. Thus programmers must manage configurations through the use of basic versioning functionality and dynamic binding. An exception is ONTOS, which provides the concept of 'configuration streams' which correspond to configuration graphs (section 4.1.2) [24] but it still has quite rigid versioning semantics. None of these offers the transparency provided by OVJ, so none are able to fill the same role of hiding object versioning management complexity from the application programmer.

The concept of 'database versions' in multi-version databases [25] is perhaps the closest in the literature to our notion of transparency. A multi-version database may be seen as a generalization of a conventional (mono-version) database, each database version being equivalent to a conventional database. The language DML [25] includes the concept of a 'default database version' (configuration) which is similar to our concept of an 'implicit configuration context' in OVJ, but DML is limited to a query language.

5.3 Directions for Future Work

The work presented here directly addresses some of the major issues facing implementers of high performance object server technology. Nonetheless many challenges remain. Perhaps the most pressing of these is schema evolution and schema versioning. While we have discussed at some length the motivation for and implementation of transparent *data* (object instance) versioning (realized as OVJ), schema versioning concerns *meta-data* (class) versioning. This is very important because just as data changes over time, so too does meta-data—both because systems modeled by the meta-data change and also because the way systems are modeled changes (improvements in algorithms, bug fixes, etc). Thus different historical instances of a 'single' object may have been created by different classes, according to when they were created.

Although we have not reported here our work on schema versioning, we have been actively researching this area and are in the process of implementing transparent support for schema versioning which we will employ in the ABS-BR demonstrator. We are also working on a number of other topics related to the ABS-BR, including extending OPJ to applets.

6 Acknowledgments

The work reported in this paper has been conducted over an eighteen month period by the members of the UPSIDE research group of the Cooperative Research Center for Advanced Computational Systems (ACSys). The authors would particularly like to thank John Zigman, Gavin Mercer, Luke Kirby, and Zhen He for their part in the development of PSI, OPJ, OVJ and ABS-BR demonstrator.

7 Conclusion

Object server technologies are becoming increasingly important, however large applications—such as our case study, the ABS-BR—are being bogged down by *complexity* and are struggling to *scale*. The task of implementing the application's business logic is clouded by the need to manage concerns such as persistence, transactions, versions, heterogeneity, distribution, etc. The consequence of this is complex code, large applications, long development times, and high maintenance costs.

We have used the ABS-BR case study as a forum for examining these problems and platform for evaluating new solutions. The approach we have presented here is deeply rooted in the principle that *abstraction* is the key in the fight against software complexity, and has been shaped by our use of Java as a development context. In our implementations of orthogonally persistent Java (OPJ) and orthogonal versioning for Java (OVJ) we have shown how effectively this principle of abstraction can be applied to the problem at hand, and demonstrated the utility of Java as a context for implementing such solutions.

While the commercial viability of such a pervasive application of the principle of abstraction has at times been questioned [10], we view the emergence of Java as a turning point in that quest. To us, the cleanness, efficiency, generality and portability of the solutions we have developed are indicators that the approach is on the cusp of being commercially viable. The progress of the new Java Data Objects (JDO) standard suggests to us that industry is beginning to think so too.

References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *OOPSLA'97, Proceedings on the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications, Atlanta, GA, U.S.A., October 5–9, 1997*, volume 32 of *SIGPLAN Notices*, pages 49–65. ACM, October 1997.

[2] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a compiler-supported Java Virtual Machine for servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99), May 1, 1999, Atlanta, Georgia, USA*, 1999.

[3] 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 Richard Connor and Scott Nettles, editors, *Seventh International Workshop on Persistent Object Systems, Cape May, NJ, U.S.A., May 33–47, 1996*. Morgan Kaufmann, 1997.

[4] Malcolm P. Atkinson and Ronald Morrison. Orthogonally persistent systems. *The VLDB Journal*, 4(3):319–402, July 1995.

[5] Douglas Barry and Torsten Stanienda. Solving the Java object storage problem. *Computer*, 31(11):33–40, November 1998.

[6] Stephen M Blackburn. *Persistent Store Interface: A foundation for scalable persistent system design*. PhD thesis, Australian National University, Canberra, Australia, August 1998. Available online at `http://cs.anu.edu.au/~Steve.Blackburn/`.

[7] Stephen M Blackburn and John N Zigman. Concurrency—The fly in the ointment? In Ronald Morrison, Mick Jordan, and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Third International Workshop on Persistence and Java, September 1–3, 1998, Tiburon, CA, U.S.A.*, San Francisco, 1998. Morgan Kaufmann.

[8] Boris Bokowski and Markus Dahm. Poor man's genericity for Java. In *Proceedings of JIT'98, Frankfurt am Main Germany, November 12–13, 1998*. Springer Verlag, 1998.

[9] Michael J Carey, David J. De Witt, and Jeffrey F. Naughton. The OO7 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data, Washington D.C., U.S.A., May 26–28*, volume 22 of *SIGMOD Record*, pages 12–21. ACM, 1993.

[10] Michael J. Carey and David J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of the 22th International Conference on Very Large Data Bases, Mumbai, India, September 3–6, 1996*, pages 3–14. Morgan Kaufmann, 1996.

[11] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, and Seth J. White. Shoring up persistent applications. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings on the 1994 ACM-SIGMOD Conference on the Management of Data, Minneapolis, MN, U.S.A., May 24–27, 1994*, volume 23 of *SIGMOD Record*, pages 383–394. ACM, 1994.

[12] Roderic Geoffrey Galton Cattell and Douglas K. Barry, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufman, December 1999. This reference must be checked before publication.

[13] GemStone Systems. GemStone/J. http://www.gemstone.com/, 1999.

[14] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[15] Matthew Hohlfeld and Bennet Yee. How to migrate agents. http://www.cs.ucsd.edu/users/bsy/pub/migrate.ps, August 1998.

[16] Antony Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahnmath. Optimizing the read and write barriers for orthogonal persistence. In Ronald Morrison, Mick Jordan, and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, Tiburon, CA, U.S.A., August 30–September 1, 1998*, pages 37–50, San Francisco, 1998. Morgan Kaufmann.

[17] Gökhan Kutlu and J. Eliot. B. Moss. Exploiting reflection to add persistence and query optimization to a statically typed object-oriented language. In Ronald Morrison and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, Tiburon, CA, U.S.A., August 30–September 1, 1998*, pages 123–135, San Francisco, 1998. Morgan Kaufmann.

[18] San-Won Lee and Hyoung-Joo Kim. Object versioning in an odmg-compliant object database system. *Software—Practice and Experience*, 5(29):479–500, 1999.

[19] Daniel Lieuwen and Narain Gehani. Versions in ode: Implementation and experiences. *Software—Practice and Experience*, 5(29):397–416, 1999.

[20] Simon Lippman. *The C++ Primer*. Addison-Wesley, second edition, 1991.

[21] Alonso Marquez, John N Zigman, and Stephen M Blackburn. Fast portable Orthogonally Persistent Java. *Software—Practice and Experience*, 2000. (to appear).

[22] Florian Matthes, Rainer Müller, and Joachim W. Schmidt. Towards a unified model of untyped object stores: Experiences with the Tycoon Store Protocol. In *Advances in Databases and Information Systems (ADBIS'96), Proceedings of the Third International Workshop of the Moscow ACM SIGMOD Chapter*, 1996.

[23] David S. Munro, Richard C.H. Connor, Ronald Morrison, Stephan Scheuerl, and David W. Stemple. Concurrent shadow paging in the Flask architecture. In Malcolm Atkinson, Véronique Benzaken, and David Maier, editors, *Sixth International Workshop on Persistent Object Systems, Tarascon, France, September 5–9*, Workshops in Computing Series (WICS). Springer-Verlag, 1994.

[24] ONTOS. *ONTOS DB 3.1 Versioning Guide*. 1995.

[25] Genevieve Jomier Stephane Gancarski and M. Zamfiroiu. A framework for the manipulation of a multiversion database. In A. Min Tjoa Norman Revell, editor, *DEXA Workshop 1995, London, United Kingdom*, pages 444–471. ONMIPRESS, San Mateo, California, 1995.

[26] Sun Microsystems. JavaBeans. API specification, Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043, 24 July 1997.

[27] Sun Microsystems. Java Data Objects Specification, JSR-12. http://java.sun.com/aboutjava/communityprocess/jsr, Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043, July 1999.