Realizing High Performance Orthogonal Persistence

Steve Blackburn and Alonso Marquez*

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia

Email: {Steve.Blackburn,Alonso.Marquez} @cs.anu.edu.au

Abstract

A rapidly growing demand for networked information has stressed the need for effective ways of managing large amounts of complex data. While more traditional solutions such database technology are already playing a major role, it seems clear that the mismatch between the relational and object oriented paradigms will be exposed by some applications as a source of major inefficiency.

The upside project at the ANU has been investigating the application of orthogonal persistence to large information servers—what we refer to as ‘high performance orthogonal persistence’. Central to our work has been exposure to a major application that seems well suited to such technology, the Australian Bureau of Statistics’ Business Register project.

This paper discusses our approach to high performance orthogonal persistence, outlines some of our achievements to this end, and looks at some of the key problems that remain.

1 Introduction

The convergence of computing and communications has had a major impact on the computer science landscape. One consequence is a rapidly growing demand for networked information, which has stressed the importance of developing scalable means of managing persistent data. While much commercial attention is focussed on relational and object-relational approaches to persistent data management, it seems to be increasingly clear that the mismatch between the relational paradigm and object orientation will be insurmountable for some classes of problem.

1.1 ABS Business Register Project

For the past year the upside project at the ANU has been involved in a collaborative project with the Australian Bureau of Statistics (ABS) investigating alternative technologies for managing large complex data sets such as those seen in the ABS Business Register. The project seeks on one hand to provide ABS with an insight into future technologies for the management of persistent data, and on the other to provide the upside group with an insight into the demands of ‘real-world’ persistent application systems1.

The Business Register (BR) is a repository of information on all of Australia’s businesses, from corner stores to multinationals. The BR aims to model the structure of, and relationships between each of the businesses. The BR was originally built as a relational database application, but has been recently moved to an object oriented database platform. The application is very large, with over one hundred classes, more than one hundred million objects and tens of gigabytes of data. Data is constantly updated, both through on-line data entry and batch processing. As a central resource for the Bureau’s economic surveys, the BR is very much ‘mission critical’.

The key challenges presented by the BR application are scalability and issues of programmability associated with the management of complex data. The main scalability demands are in terms of: communications (being able to support a large number of distributed clients concurrently); computation (having sufficient computational resources

---

1 The authors wish to acknowledge that this work was carried out within the Cooperative Research Centre for Advanced Computational Systems established under the Australian Government’s Cooperative Research Centres Program.

2 It should be noted that strict privacy legislation prevents the ABS from disclosing any live BR data to us—we therefore work with synthetic data based on the BR schema and population characteristics.
to respond to requests in a timely manner); and storage (having sufficient primary and secondary storage and storage bandwidth available). The programmability challenge is seen in the need to be able to efficiently maintain (and in the first instance, develop) the persistent application software in the face of a number of critical and complex demands such as: persistent data management (in an OODB setting this involves maintaining mappings between persistent and non-persistent data); access to temporal instance data (viewing the registry in terms of its state at various times in the past); statistical integrity constraints (statistical factors such as ‘dependent source feedback’ require the provision of multiple views of the data); and schema versioning/evolution (as a model of real-world businesses, the BR schema must reflect changes in business structures such as those induced by legislative change).

Given our research background in the areas of scalability and orthogonal persistence, the upside group has approached the ABS BR problem as a potential application for high performance orthogonal persistence (HPOP). The remainder of this paper discusses our efforts to realize HPOP, first by defining what we mean by HPOP, and then by describing our approach to delivering HPOP.

2 High Performance Orthogonal Persistence

We use the term ‘high performance orthogonal persistence’ to refer to the application of orthogonal persistence to data management contexts where performance demands in terms of communications, computation, storage, and storage bandwidth are high. While such a context might seem natural from the perspective of a database researcher, it seems clear that there are other important application domains for orthogonal persistence with quite different characteristics [Atkinson and Morrison 1995].

Given this definition of HPOP, the following appear to be essential criteria for a successful realization of HPOP:

- A scalable architecture (with respect to communications, computation, storage and storage bandwidth),
- A scalable programming model (that will allow the effective utilization of a scalable architecture),
- A usable programming environment (with respect to issues such as schema change, concurrency etc.),
- An efficient programming environment (with respect to performance standards of database systems).

The following sections address each of these concerns and our response to them.

3 Scalable Architectures

Previous work at the ANU has lead to the identification of the transactional object cache architecture as a foundation for scalable persistent system construction [Blackburn and Stanton 1997; Blackburn 1998]. The fundamental concepts underpinning the transactional object cache (TOC) framework are caching, atomicity, and layering. Key properties of the architecture include: a separation of store and language concerns; transparency (to the user) of store implementation issues such as distribution; and the use of atomicity to hide latencies. The TOC framework has been fleshed out in the definition of PSI, the Persistent Store Interface, which concretely identifies the interface between store and programming language concerns.

There are a number of implementation challenges associated with the realization of a scalable store, most notably the availability of scalable underlying hardware and the development of scalable algorithms for cache coherency, recovery, and garbage collection. The ready availability of SPM hardware platforms that scale to many tens of processors and MPP hardware that scales into the hundreds and even thousands leaves the development of scalable algorithms as the key issue. The scalability of transactional cache coherency algorithms and recovery algorithms has been experimentally demonstrated [Blackburn 1998; Blackburn et al. 1997]. The development of scalable, transactional garbage collection is the subject of current investigation.

4 Scalable Programming Models

Given a scalable architecture for persistence, the problem of identifying a programming model that can effectively utilize the architecture remains. In the world of scalable computing, two landmarks on the spectrum of programming models are often characterized as the throughput and response time orientations. The orientations can be thought of in terms of a possible choice between high bandwidth and low latency respectively. Looking beyond the stopwatch towards the underlying question of programming models, it seems that the key issue lies in a choice of orientations
between isolation or cooperation (or perhaps locality and dispersion). It is clear that in a context where communications is the key performance bottle-neck, isolation (locality) will always beat cooperation (dispersion) in the race for performance. The choice of programming model for a scalable context is therefore weighted heavily in favor of isolation over cooperation.

Given the abovementioned advantages of isolation and the broad application domain identified for HPOP, a transactional programming model is a natural first choice for HPOP. However, this raises serious problems—first the limitations that isolation bring to a programming model, and second the difficulty of integrating transactions into orthogonal persistence. The response to the first problem lies in the development of flexible transaction models—a task that will be helped by the imperative of ‘real world’ problems, such as those we are beginning to see through the ABS BR project. The second problem is non-trivial, although the chain-and-spawn transaction model offers a simple (and somewhat limited) escape [Blackburn and Zigman 1998].

5 Orthogonally Persistent Java

Having addressed the first two criteria for realizing HPOP, we come to the last two—the usability and efficiency of the programming environment. The starting point is to select a programming language to work with. Given the commercial pragmatics of our project and the relative suitability of Java to orthogonal persistence, our initial choice has been to select an implementation of orthogonally persistent Java (OPJ) as the basis of our programming environment. In selecting an implementation, the following criteria were of particular importance:

- Cleanliness of the model of persistence presented,
- Support for transactions,
- Capacity to accommodate our store and language level research,
- Performance.

None of the currently available OPJ implementations were able to satisfy these criteria, and so we have developed our own OPJ prototype.

5.1 A Poor Man’s OPJ (PM-OPJ)

It is not within the scope of this paper to describe PM-OPJ in great detail. Instead, an overview of the implementation and a brief account of a number of its characteristics will be given.

PM-OPJ is a ‘pure-Java’ implementation insofar as it does not depend on modifications to the virtual machine (VM), or compiler. Instead PM-OPJ uses a customized class loader which translates bytecodes at load time, inserting read and write barriers as necessary. This gives the implementation great portability as it will (in principle) run on any Java VM.

Static Variables as Roots  PM-OPJ does not require the identification of explicit persistent roots. Instead it uses the static variables of classes as implicit roots of persistence. The store is invoked by executing Java (using the PM-OPJ classloader) with respect to a class with a main() defined. If the class is in the store then it will be executed and will see all data reachable from its static variables. The cleanness of this model was made clear when the OO7 benchmark was ported from PJama[Atkinson et al. 1996] to PM-OPJ. The port simply required commenting out all references to PJama’s PJStore class (such as getStore(), newPRoot(), etc.).

Purely Transactional Orthogonal Persistence  PM-OPJ utilizes the ‘chain and spawn’ transaction model [Blackburn and Zigman 1998] to provide a purely transactional orthogonal persistence. All computation exists within a transactional context and inter-transactional concurrency is facilitated by the use of chain and spawn, which allows new transactional threads to be thrown from within a transaction.

PM-OPJ does not explicitly enforce transactional isolation, instead its guarantees of isolation rely on those of PSI and Java’s classloader name-space isolation mechanism. This obviates the need to implement any form of locking at the language level (and any consequent performance penalties). Furthermore, PM-OPJ’s use of PSI allows it to exploit any store architecture that supports the PSI interface—PM-OPJ could therefore be trivially applied to a multicomputer context contingent only on the existence of a PSI-compliant store for that architecture.
**Bytecode Transformations**  Read and write barriers—the core of any orthogonal persistence implementation—are implemented in PM-OPJ by bytecode transformations that occur at class loading time. The transformation process need only happen once, as persistent classes retain the transformations.

Orthogonally persistent applets and transparent semantics such as instance and schema versioning depend on the same bytecode transformation technology. A critical aspect of the implementation of these features is the efficient use of ‘proxy objects’—something that is made possible by optimizations carried out within the bytecode transformation process.

### 6 Future Work

Despite the progress towards realizing HPOP that is documented in the preceding pages, there remains much to be done.

The implementation of scalable stores for the support of HPOP is nearing completion. The primary challenge that remains is the development of scalable, transactional garbage collection technology. The development of algorithms to this end is a high priority for the upside group.

PM-OPJ is a first prototype, developed over a very short period. Much can be done to optimize its performance and much work remains in completing the PM-OPJ programming environment to the point that transparent instance and schema versioning are supported. We will continue to investigate alternative approaches to building OPJ, including those involving VM and compiler modifications.

The final area that needs to be addressed is in the search for better programming models for HPOP. The limitations of simple transaction models seem clear, as does the motivation for identifying practical alternatives for the HPOP setting.

### 7 Conclusions

There is a clear demand for high performance orthogonal persistence. The ABS BR project has provided us with clear motivation by way of challenging ‘real world’ problems. We have responded to these challenges with a pragmatic vision for HPOP and have made major progress in realizing that vision. Despite this progress clear problems remain, both at the technical level—as in the search for transactional, distributed garbage collection—and at higher levels—as in the search for more appropriate programming models.

### Bibliography


