*Proceedings of the Second International Workshop on Persistence and Java*, Mick Jordan and Malcolm Atkinson (Eds.)
SMLI TR-97-63, Sun Microsystems Laboratories, Dec. 1997

# Analysing, Profiling and Optimising Orthogonal Persistence for Java[*]

Position Paper for the
Second International Workshop on Persistence and Java
California, August 1997

Quintin Cutts
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland
quintin@dcs.gla.ac.uk

Antony L. Hosking
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
hosking@cs.purdue.edu

August 22, 1997

**Abstract**

Persistent systems manage main memory as a cache for efficient access to frequently-accessed persistent data. Good cache management requires some knowledge of the semantics of the applications running against it. We are attacking the performance problems of persistence for Java through analysis, profiling, and optimisation of Java classes and methods executing in an orthogonally persistent setting. Knowledge of application behaviour is derived through analysis and profiling, and applied by both a static bytecode transformer and the run-time system to optimise the actions of Java programs as they execute against persistent storage. Our prototype will unify distinct persistence optimisations within a single optimisation framework, deriving its power from treatment of the entire persistent application, consisting of both program code and data stored in the database, for *whole-application* analysis, profiling and optimisation.

**Keywords:** persistence, Java, bytecode, program analysis, dynamic profiling, optimisation

## 1 Introduction

Orthogonally persistent programming languages [Atkinson and Buneman 1987; Atkinson and Morrison 1995] provide improved support for the design, construction, maintenance and operation of applications that manage large bodies of long-lived, shared, structured data. In spite of this, there is continued mainstream resistance to languages with orthogonal persistence due to a perception that they cannot deliver performance to match that of traditional programming languages. We believe that performance problems associated with persistence can be dealt with through extension of traditional program analysis and optimisation techniques to encompass optimisation of persistent programs, as well as new techniques based on execution profile feedback [Hölzle and Ungar 1994; Grove et al. 1995], specialisation, customisation and other partial evaluation [Chambers and Ungar 1989; Chambers et al. 1989; Chambers and Ungar 1990; Chambers 1992; Dean et al. 1995; Dean et al. 1995; Consel and Danvy 1993; Jones et al. 1993], and dependence analysis and loop restructuring [Wolfe 1996]. We also believe that analysing, profiling and optimising in a persistent setting has benefits even for generic optimisations not directly related to persistence.

### 1.1 Orthogonal persistence

The language principles of *transparency* and *orthogonality* have been repeatedly articulated [Atkinson and Morrison 1995; Moss and Hosking 1996] as important in the design of persistent programming languages, enabling the full

[*]Java is a trademark of Sun Microsystems, Inc.

power of the persistence abstraction. Transparency means that access to persistent objects does not require explicit calls to transfer them between stable store and main memory. Thus, a program that manipulates persistent (or potentially persistent) objects looks similar to a program concerned only with transient objects. To support this, the program's compiled code or interpreter, and the persistent run-time system, contrive to make objects resident in memory on demand, much as non-resident pages are automatically made resident by a paged virtual memory system.

Treating persistence as *orthogonal* to type encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax and store semantics. Thus, programmers need to add little to their understanding of the language in order to begin writing persistent programs. A common way to achieve orthogonal persistence is by treating persistent storage as a stable extension of the dynamic allocation heap. This allows a uniform and transparent treatment of both transient and persistent data; persistence is orthogonal to the way in which objects are defined (i.e., their types), allocated, and manipulated in the heap.

Implementation of transparent, orthogonal persistence requires two underlying mechanisms: *residency checks* to trigger retrieval of non-resident objects from stable store to main memory for read access, and *update checks* to track their modifications (including acquisition of write locks as necessary for concurrency control) for eventual propagation back to stable storage. The application of these checks constitutes a *read barrier* and *write barrier* for persistence, respectively, since execution is suspended until the checks and any consequent action is complete. Efficient implementation of these mechanisms, and sensible object caching policies, are the keys to performance for persistence.

## 1.2 Performance

Cattell [1994] (p. 268) mentions two performance tenets for an object data management system (ODMS):

**T26:** "*Minimal access overhead.* An ODMS must minimise overhead for simple data operations, such as fetching a single object."

**T27:** "*Main-memory utilisation.* An ODMS must maximise the likelihood that data will be found in main memory when accessed. At a minimum, it should provide a cache of data in the application virtual memory, and the ability to cluster data on pages or segments fetched from the disk."

We are addressing both of these issues, which can significantly affect performance: reducing access overhead leads to persistent programs whose performance approaches that of their non-persistent counterpart, since the persistent program will have negligible overhead when operating entirely on memory-resident data; improving main-memory utilisation reduces I/O which is the main performance barrier for persistence.

### *1.2.1 Minimal access overhead*

There is an inherent tension between the principle of orthogonality and efficient implementation of that model. The abstraction of orthogonal persistence obscures the performance disparity between fast cache/main memory and slow secondary storage. Thus, naive implementations of orthogonal language designs can lead to inefficiencies. For example, a given object reference in an orthogonal persistent program may target either a resident or non-resident object. Before the object can be manipulated through that reference a residency check must be performed to make sure the object is available in memory. A naive implementation would encode each object reference using its target object's disk-based persistent identifier (PID). Every time an object reference is traversed, the PID must be mapped to a pointer to the target object in memory (with a call to make the object resident if it is not already). Residency checks on transient or already-resident persistent objects are unnecessary, so long as those objects remain resident. Eliminating the check and using a direct memory pointer to refer to such objects is more efficient, since repeated object access can be achieved through fast main-memory addressing as opposed to slow PID translation. We are developing global data flow analyses (both intra- and inter-procedural) to discover and eliminate redundant residency and update checks and to influence conversion of PIDs to direct pointers (i.e., "swizzling").

### *1.2.2 Main-memory utilisation*

Persistence transparency precludes explicit control by the programmer over the physical transfer of objects between main memory and persistent storage. Rather, objects are automatically retrieved as needed by the program and cached in memory until evicted by the cache replacement policy. However, I/O latencies are so high that the timing of fetch requests, the way in which objects are clustered for storage and retrieval, and the policy for object replacement all have a significant impact on the performance of a persistent program.

Note that persistent systems face a much more difficult task of cache management than file-based systems. A persistent store contains highly-structured complex objects that are traversed in relatively random orders with respect to their storage on disk, rendering the low-level clustering and prefetching strategies of traditional file systems ineffective.

We are devising techniques for automatic derivation of strategies for prefetching, replacement, and clustering of objects, based on static program analysis, dynamic profiling, and direct consideration of the schema and physical structure of the target database.

## 2 Optimisation

Our goal is to devise, implement, and evaluate optimisations for the orthogonal persistence for Java (OPJ) prototype [Atkinson et al. 1997; Atkinson et al. 1996] being developed jointly by Sun Laboratories and the University of Glasgow. We divide candidate optimisations into those that are:

**enabled by** persistence: optimisations that target generic program improvement and are enabled by analysis, compilation and execution in a persistent setting; and

**enabling for** persistence: optimisations specifically targeting reduction of the persistence-specific overheads of execution

We now describe in more detail the optimisation strategies we plan to explore.

### 2.1 Persistence-enabled optimisations

Java's dynamic, late-binding, object-oriented nature provides many opportunities for optimisation, such as those pursued for Self [Ungar and Smith 1987] and other OO languages: e.g., specialisation, customisation, method splitting, cloning and inlining, which all act to reduce the overhead of dynamic method invocations. Similar optimisations have been applied in other settings such as Modula-3, based on ”whole-program” analysis [Fernandez 1995; Diwan et al. 1996; Diwan 1997].

Java's model of network code distribution dictates a standard class file format for network transmission, with code taking the form of architecture-neutral bytecode instructions for the Java Virtual Machine (VM) [Lindholm and Yellin 1996]. In such a setting Java source code is never transmitted to clients: they see only the VM bytecodes, which they may interpret with a virtual machine, or translate to native code using a ”just-in-time” (JIT) compiler for direct execution. Without any control over the initial server-side compilation phase that translates source code to bytecodes, the only opportunity for clients to influence code quality is at the time of JIT compilation after receipt of the Java bytecodes. However, there is an inherent tradeoff to optimisation during JIT compilation, since the goal is usually to begin executing code as soon as possible after it has been received, whereas many optimisations depend on a time-consuming analysis of the code. Indeed, Self-93 [Hölzle and Ungar 1996] retreated from many of the more expensive analyses and optimisations of Self-91 [Chambers 1992] simply because they were too expensive for fast turnaround of JIT compilations.

It is here that persistence can provide assistance, allowing more aggressive off-line analyses and optimisations of both bytecode and native code. Analysis and optimisation phases are greatly simplified when all code, data, profiles and other measurements are retained within a persistent store. Advantages accrue from the accumulation of large bodies of types, analysis results and both unoptimised and optimised bytecode and native code in the persistent store. JIT compilers can take advantage of such analysis to generate more efficient code, but without the overhead of on-line analysis. In this sense, persistence changes the game because there is no longer a need to trade off performance for responsiveness.

OPJ calls for Java classes (including code) to be interned as part of the persistent store. When loading a Java class, OPJ first checks to see if the class is already available in persistent storage; if so then the interned code can be used instead of loading the external bytecoded code representation. Needless to say, interned persistent code can take whatever form is convenient: standard bytecodes, extended bytecodes, or native code. Whatever form is assumed for interned code, the expensive analyses that drive candidate optimisations can be performed on the stored code during periods when the persistent system is inactive. Moreover, the interned body of code can approximate the notion of ”whole-program” that has been exploited for optimisation in other settings. Naturally, open-world assumptions must apply where necessary, either to avoid recompilation of vast quantities of code when new classes are interned and existing classes are modified, or to trigger re-optimisation and re-compilation if appropriate. Such interaction between optimisation and system evolution is a promising area for further investigation, as is the concept of “active” for a persistent system.

Another interesting twist is that in a persistent setting code can be specialised with respect to the stored data (both the instances and the instantiated classes). Thus we are presented with a classic opportunity for partial evaluation of code with respect to the database schema, its physical structure, and the stored instances. The schema consists of the set of types of all objects actually stored in the database, as opposed to described in the code. Where there are differences between the types described in the code and the database schema we have an opportunity for optimisation based on type hierarchy analysis [Dean et al. 1995; Diwan et al. 1996; Diwan 1997], which bounds the set of procedures a method invocation may call by examining the object-oriented type hierarchy for method overrides. It is these overrides that result in dynamic method invocations at so-called polymorphic call sites. Although the program’s type hierarchy may imply a polymorphic call is necessary, the database schema’s hierarchy might indicate that only a monomorphic call is required, since objects of only one of the types possible in the polymorphic call can actually be allocated or encountered in the database. Thus, the indirect polymorphic call can be converted to a direct monomorphic call.

Specific to Java there are remaining interesting problems for optimisation that also arise out of its inherent dynamism, where the body of code in the system can evolve over time. On the one hand, persistence helps by enabling complex, long-running analyses in a dynamic object-oriented environment, while on the other hand, static analysis can become obsolete at any instant. It seems that this is very different from prior settings for optimisation.

## 2.2 Persistence-enabling optimisations

These are our primary focus. Whereas persistence-enabled optimisations take and extend to a persistent setting previous work in the area of optimisation of non-persistent aspects of execution, persistence-enabling optimisations directly address the performance of persistence features. Persistence optimisations strive for minimal access overhead for persistent data and improved main-memory utilisation as described above. Minimising access overhead means that simple operations can be performed on persistent data with minimal overhead (ideally, there should be little or no performance penalty for manipulating resident persistent data compared to transient data). Good main-memory utilisation means maximising the likelihood that data will be found in memory when accessed, through caching of data in the application virtual memory and clustering of related data on pages or segments fetched from disk.

Persistence optimisations are driven by information about the *co-residency* of particular objects. We write $i \rightarrow_p j$ to indicate that whenever an object $i$ is resident so also $j$ will be resident, with probability $p$. We write $i \rightarrow j$ if $p = 1$. If $i \rightarrow j$ and $i$ is made resident then references to $j$ can be swizzled to direct memory pointers. Residency check elimination assumes that the run-time system will respect co-residency assumptions. By default, we assume $i \rightarrow i$

(once resident an object will stay resident so long as “live” swizzled pointers to it exist). Thus, residency checks are idempotent, and redundant checks can be eliminated. Further, given $i \rightarrow j$, references from $i$ to $j$ can be traversed without checks. Similarly, update checks and lock acquisition for transaction concurrency control are idempotent within transaction boundaries, and can be optimised in similar fashion.

Co-residency is transitive only if all weights $p$ have value 1. One can think of adjusting the co-resident reach of a given “handle” on a persistent data structure by combining weights and applying a threshold. Suppose $i \rightarrow_p j \rightarrow_q k \rightarrow_r l$, specifying that $j$ should be co-resident with $i$ with weight $p$, $k$ with $j$ with weight $q$, and $l$ with $k$ with weight $r$. Assume $0 \leq p, q, r \leq 1$. Then, given a ”handle” on $i$, and some threshold $t$, $j$ will be co-resident if $p > t$, $k$ if $pq > t$ and $l$ if $pqr > t$. That way, a given handle can modulate its ”reach” using the threshold combined with the weights on the edges of the data structure.

Note that when executing code that is compiled to take advantage of co-residency assumptions the object cache manager is required to guarantee residency of certain objects. In a multi-threaded environment, these guarantees require careful management to avoid severe performance degradation. Section 2.4 considers this issue in more detail.

Swizzling can also be driven by co-residency information. If $i \rightarrow j$ holds then we might as well swizzle references to $j$ contained in $i$. Not only are checks on those references redundant, but the link can be followed with minimal overhead. Prefetching and clustering can be driven similarly: when fetching $i$ we might as well issue a request for (i.e., prefetch) $j$ at the same time; if $j$ is also clustered with $i$ then further I/O is unnecessary.

As implied earlier, co-residency information can be acquired in several ways. Static data-flow analysis can approximate the ”storage profile” of a piece of code which can be refined through dynamic profiling. The information might be encoded as constraints that must be obeyed by the run-time system before the code (compiled in light of the constraints) can execute, or more globally, a collection of related types can be annotated by the system to indicate the global storage profile of their instances, with code optimised in light of those global annotations [Moss and Hosking 1995]. Other static residency information can be gleaned from knowledge of the object-oriented execution paradigm of Java: the target of any dynamic method invocation (i.e., the “this” argument) must be resident in order to dispatch the method. Thus, the bodies of those methods can access the fields of the target object without residency checks. This observation led to elimination of 86-99% of residency checks in a prototype persistent Smalltalk system [Hosking 1997]; we expect similar improvements for OPJ programs.

To sum up, the principal objective is to focus on the unique setting persistence gives for optimisation, both persistence-enabled and persistence-enabling. There is a strong connection among persistence optimisations such as residency-/update-/lock-check elimination, clustering, prefetching, and swizzling, centered on co-residency analyses and dynamic profiling. Prior work in these areas has not recognised the commonalities that arise in each of them, and our goal is to unify the approaches in a common framework of program analysis and execution profiling. Recasting previous optimisations (e.g., from Self) in this framework is a by-product of this objective.

### 2.3 Analysis and optimisation framework

In order to understand our approach it is necessary to consider the current architecture of OPJ. Without changing the standard Java language, the OPJ team has made extensions to JavaSoft’s Java Development Kit (JDK) implementation of the VM to support transparent, reachability-based persistence for Java. Stable storage is currently provided as a layer below the standard JDK VM and its garbage-collected volatile heap. Attempts to access non-resident objects are trapped by the OPJ VM. The resulting object faults are serviced by calls to the storage layer to fetch, swizzle as necessary, and cache a copy of the target object in memory, before execution can proceed.

We are focusing on persistence optimisations, but intend also to consider the synergies between persistence and generic optimisations, particularly those able to take advantage of analysis and profile information stored along with the persistent store schema (e.g., the instantiated types in the store) and the store’s physical organisation (e.g., clustering, indexes, etc.).

We plan to analyse and optimise to an extended VM bytecode set from the standard bytecodes. Some simple analyses and static optimisations might be performed by an extended VM itself when the code is interned (e.g., replacing slow bytecodes with their quick forms as the current JDK VM already does), but more complicated analyses and transformations will take place off-line. Short of native-code compilation there is much we can do to improve the performance of OPJ. We will modify the OPJ VM to incorporate additional non-standard bytecodes for use in optimised interned code. These bytecodes will isolate and expose the costlier operations of persistence. Static data-flow analysis and optimisation, allied with execution profile feedback, will determine where these bytecodes are redundant and so can be eliminated.

As a concrete example, consider residency checking. Currently, OPJ residency checks are performed on every ”unhandle” operation in the VM. By removing the checks from the internal unhandle operation and exposing them as separate bytecodes we can eliminate those found to be redundant. (Similar strategies can be applied for clustering, prefetching, and elimination of unnecessary update checks and lock requests [Hosking and Moss 1991; Moss and Hosking 1995; Hosking 1995; Hosking and Moss 1995; Hosking 1997].)

## 2.4 Profiling and run-time support

In addition to static analysis we will also perform significant dynamic profiling, since Java's execution allows extreme forms of dynamism (e.g., injection of classes unknown at compile-time). The results of static analysis will be both updated code sequences and auxiliary data structures capturing both analysis information and frameworks for dynamic profiling. In the persistent setting it is possible for the analyser itself to plant profiling code and attach appropriate data structures to a targeted class object to record profile information associated with that class. This is in line with previous approaches to dynamic persistence optimisations [Cutts et al. 1994].

As well as profiling localised to certain code regions, profiling of a more global nature will also be performed. For example, the cache manager, thread scheduler and lower-level store mechanisms maintain a global view of data access spanning multiple executing threads. Static code/class analysis alone cannot capture this global behaviour.

For any particular optimisation, some adjustment will be required within the OPJ VM. Particular examples are implementations for new bytecodes, and adjustments to the cache management mechanism to take into account static residency assumptions used to drive optimisation. For example, static analysis may identify redundant residency checks on the basis of guarantees about the on-going residency of the target object. That is, there is an expectation that a resident object will not be unceremoniously evicted by the cache manager. In essence, residency checks pin objects in memory for some range of code, and the cache manager must agree to maintain those objects as resident so long as the thread executing the pinning code is active.

Excessive pinning of objects will prevent effective cache management. This is especially so in the context of pre-emptive thread scheduling, where a thread could be pre-empted in the middle of a pinning range. If the pre-empted thread is pinning large amounts of data then other running threads will face a congested cache. To avoid these problems we plan to allow the cache manager to steal pinned objects from suspended threads, on the understanding that stolen pins will be restored when the thread is resumed. Analysis of the existing cache manager and thread scheduler of the OPJ VM [Daynès 1997] has shown that they can be modified to support this additional functionality efficiently. Briefly, rather than overloading execution with explicit pinning mechanism, residency checks will implicitly pin objects. So long as there exist live references from a thread stack to an object, the cache manager will avoid stealing that object, and then only if the thread itself is inactive. If it absolutely must steal from the thread then it faces two choices: make the thread inactive until such time as memory becomes available to allow the object to be pinned and the thread resumed; or arrange for access to the object to be trapped transparently with respect to the code (techniques based on operating system primitives for virtual memory page protection are one approach [Hosking and Moss 1993]). Of course, one key question is how to discern live references. Bytecode analysis can help here by capturing liveness information per code range for use at run-time, but we can assume no such information is available for native methods. Since the current object cache manager must work with native methods anyway, we see no reason why analysis-guided pinning assumptions cannot also be incorporated into its mechanisms.

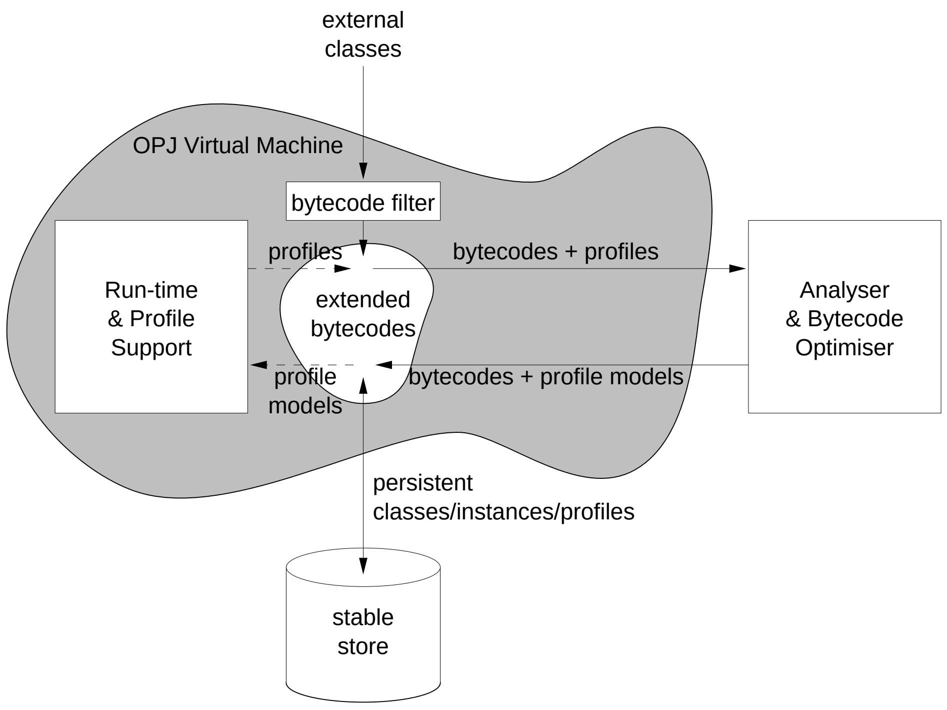

Figure 1: Prototype architecture

## 3 Prototype

We are building a prototype bytecode analyser based on the optimisation framework of Diwan [1997], and collaborating with the OPJ team to support the necessary extended bytecodes and run-time functionality to permit persistence optimisations. The basic architecture of the prototype is illustrated in Figure 1. As external classes are interned by the OPJ virtual machine, their bytecoded methods are filtered and augmented with the extended bytecodes for persistence (e.g., residency checks, update checks, etc.). These extended bytecodes simply expose persistence functionality that is currently buried inside the bytecode implementations of the current OPJ system. Assuming that the OPJ VM fully implements these extended bytecodes (i.e., as other than no-ops) then it can excise the buried persistence mechanisms and run with the extended bytecode set. The filtering process does not perform optimisation; it simply adjusts the bytecode for execution on the (extended) OPJ VM.

Analysis and optimisation takes place as necessary, on-line in response to requests by the run-time system when it discovers execution hot-spots that might benefit from optimisation, and off-line during periods of system quiescence. The analyser/optimiser is being coded in Java and will take advantage of a privileged interface to the persistent store allowing it to navigate persistent classes and to peruse and update their bytecoded methods. The analyser will communicate *profile models* to the run-time system for annotation; these consist of auxiliary data structures on which to hang execution statistics that may later serve more focused analysis and optimisation. Examples of these auxiliary structures include control-flow graphs for execution frequency annotation, and type graphs and instance structure graphs for pointer traversal annotation. Annotation may be performed by the insertion of profiling code, or through *a priori* contracts between the run-time system and the analyser. Note that all of this information can itself persist in the store for subsequent use in later executions.

## 4 Conclusion

We have briefly described a prototype bytecode analysis and profiling framework that we plan to implement for the prototype OPJ system. The intention is to support optimisations that both enable, and are enabled by, persistence. Construction of the initial prototype analyser and attendant modifications to the OPJ virtual machine for residency check elimination are expected to be nearing testing by the time of the workshop, where we will report on our experiences thus far. Areas we expect at that time to be more concrete include:

- the interface between the OPJ system and the analyser
- dynamic profiling results to expose system hotspots, and the potential benefits of candidate optimisations
- a detailed description and design of the pinning architecture within the OPJ virtual machine, including its interaction with the thread scheduler and object cache manager
- specification of the new, internal bytecodes for persistence mechanisms

## Acknowledgements

We are grateful to Amer Diwan for assistance in understanding his Modula-3 analysis and optimisation framework, on which we model our own Java bytecode analyser. For details see Diwan et al. [1996] and Diwan [1997].

## References

ATKINSON, M. P. AND BUNEMAN, O. P. 1987. Types and persistence in database programming languages. *ACM Comput. Surv. 19,* 2 (June), 105–190.

ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Record 25,* 4 (Dec.), 68–75.

ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. 1997. Design issues for persistent Java: A type-safe object-oriented, orthogonally persistent system. See Connor and Nettles [1997].

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *Int. J. Very Large Data Bases 4,* 3, 319–401.

CATTELL, R. G. G. 1994. *Object Data Management: Object-Oriented and Extended Relational Database Management Systems*. Addison-Wesley.

CHAMBERS, C. 1992. The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages. Ph.D. thesis, Stanford University.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Portland, Oregon, June). 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (White Plains, New York, June). *ACM SIGPLAN Notices 25,* 6 (June), 150–164.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, Oct.). *ACM SIGPLAN Notices 24,* 10 (Oct.), 49–70.

CONNOR, R. AND NETTLES, S., Eds. 1997. *Proceedings of the Seventh International Workshop on Persistent Object Systems* (Cape May, New Jersey, May 1996). Persistent Object Systems: Principles and Practice. Morgan Kaufmann.

CONSEL, C. AND DANVY, O. 1993. Tutorial notes on partial evaluation. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan.). 493–501.

CUTTS, Q., CONNOR, R., KIRBY, G., AND MORRISON, R. 1994. An execution driven approach to code optimisation. In *Proceedings of the 17th Australasian Computer Science Conference* (Christchurch, New Zealand). 83–92.

DAYNÈS, L. 1997. Private communication.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1995. Selective specialization for object-oriented languages. See PLDI [1995], 93–102.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming* (Åarhus, Denmark, Aug.). Lecture Notes in Computer Science, vol. 952. Springer-Verlag.

DEARLE, A., SHAW, G. M., AND ZDONIK, S. B., Eds. 1991. *Proceedings of the Fourth International Workshop on Persistent Object Systems* (Martha's Vineyard, Massachusetts, Sept. 1990). Implementing Persistent Object Bases: Principles and Practice. Morgan Kaufmann.

DIWAN, A. 1997. Understanding and improving the performance of modern programming languages. Ph.D. thesis, University of Massachusetts at Amherst.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Jose, California, Oct.). *ACM SIGPLAN Notices 31,* 10 (Oct.), 292–305.

FERNANDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. See PLDI [1995], 103–115.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C. 1995. Profile-guided receiver class prediction. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Austin, Texas, Oct.). *ACM SIGPLAN Notices 30,* 10 (Oct.), 108–123.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Orlando, Florida, June). *ACM SIGPLAN Notices 29,* 6 (June), 326–336.

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst. 18,* 4 (July), 355–400.

HOSKING, A. L. 1995. Lightweight support for fine-grained persistence on stock hardware. Ph.D. thesis, University of Massachusetts at Amherst. Available as Computer Science Technical Report 95-02.

HOSKING, A. L. 1997. Residency check elimination for object-oriented persistent languages. See Connor and Nettles [1997], 174–183.

HOSKING, A. L. AND MOSS, J. E. B. 1991. Towards compile-time optimisations for persistence. See Dearle et al. [1991], 17–27.

HOSKING, A. L. AND MOSS, J. E. B. 1993. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, Dec.). *ACM Operating Systems Review 27,* 5 (Dec.), 106–119.

HOSKING, A. L. AND MOSS, J. E. B. 1995. Lightweight write detection and checkpointing for fine-grained persistence. Tech. Rep. 95-084, Department of Computer Sciences, Purdue University. Dec.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

MOSS, J. E. B. AND HOSKING, A. L. 1995. Expressing object residency optimizations using pointer type annotations. In *Proceedings of the Sixth International Workshop on Persistent Object Systems* (Tarascon, France, Sept. 1994), M. Atkinson, D. Maier, and V. Benzaken, Eds. Workshops in Computing. Springer-Verlag, 3–15.

MOSS, J. E. B. AND HOSKING, A. L. 1996. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java* (Drymen, Scotland, Sept.), M. P. Atkinson and M. J. Jordan, Eds. Sun Microsystems.

PLDI 1995. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (La Jolla, California, June). *ACM SIGPLAN Notices 30,* 6 (June).

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, Florida, Oct.). *ACM SIGPLAN Notices 22,* 12 (Dec.), 227–241.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley.