Analysing, Profiling and Optimising for Persistence

Position Paper for the
Fourth Workshop on Integrated Data Environments – Australia
Magnetic Island, Queensland, May 1997

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

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

May 9, 1997

> "Unfortunately, little research has been reported on optimizing database programming languages. Traditional programming-language optimization techniques should be applicable in many cases, but no research has been reported … This is an important and difficult area for future research, because it spans programming language and database systems disciplines." [Cattell 1994], p. 185.

1 Introduction

Orthogonally persistent programming languages[1] provide improved support for the design, construction, maintenance and operation of applications that manage large bodies of long-lived, shared, structured data. Despite 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. The persistent languages research community has historically focused more on functionality than performance for its prototype implementations. In the development of initial prototypes such omission may be acceptable, but with persistence abstractions now reasonably well understood it is time to focus on delivering performance as well. We believe that performance problems associated with persistence can be dealt with through extension of traditional techniques for compile-time analysis and optimization of persistent programs, as well as new techniques based on execution profile feedback,[2] specialization, customization and other partial evaluation,[3] and dependence analysis and loop restructuring.[4] Profiling and optimising in a persistent setting also has potential benefits for generic optimisations not directly related to persistence.

We are developing language-independent techniques for the elimination of both CPU and I/O overheads of persistent programs, enabled by contractual obligations between the optimizing compiler and persistent run-time system. The techniques are being implemented for our own persistent extension of Modula-3 [Nelson 1991] and the Sun/Glasgow orthogonal persistence for Java implementation [Atkinson et al. 1997; Atkinson et al. 1996]. Our work will also consider how standard optimizations for object-oriented languages can be integrated within a persistent framework.

[1] [Atkinson and Buneman 1987; Atkinson and Morrison 1995]

[2] [Hölzle and Ungar 1994; Grove et al. 1995]

[3] [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]

[4] [Wolfe 1996]

2 Performance

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

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

T27: "*Main-memory utilization*. An ODMS must maximize 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."

Our work addresses 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 utilization reduces I/O which is the main performance barrier for persistence.

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. The circumstances in which residency checks can be eliminated are outlined in more detail below.

Main-memory utilization: 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. We are also 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.

3 Optimisations

Our approach is language-oriented, with the language compiler/analyser *and* run-time system taking joint responsibility for persistence. This differs from approaches that rely solely on the operating system or complex underlying database system to support persistence, where language-level semantic information is lost before they are aware of the program. Rather, we enables opportunities for more efficient persistent programs by attacking performance problems through compiler/run-time cooperation. Persistent object management will be tailored to the specific semantics of the language, individual application programs, and the schema (i.e., the types) and physical structure of the target database. Together, the compiler and run-time system

will present the abstraction of persistent storage while ensuring performance through coordinated, automatic management of the storage hierarchy for application programs.

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, as inactive threads may pin large amounts of data.

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 and Modula-3: 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.

4 Perspective

Taking a broader perspective, persistence is one form of *late binding*: it allows augmentation of a running environment with object instances (and their classes) that have been stored in a database during previous executions. In a persistent setting not only are classes (and their code) dynamically bound, but so also are object instances, requiring dynamic resolution of references to persistent objects before they can be manipulated. It is this dynamic binding that introduces the overhead (both CPU and I/O) that we hope to optimize away. Late binding is an increasingly common attribute of many modern programming languages, supporting language features that offer flexibility to the programmer and simplify the task of programming, such as object-orientation and dynamic loading of new code. How persistence and optimizations for persistence in-

teract with optimizations for these other instances of late binding is one direction in which the work outlined here will be extended.

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.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1995. Selective specialization for object-oriented languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (La Jolla, California, June). *ACM SIGPLAN Notices 30,* 6 (June), 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.

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.

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

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

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.

NELSON, G., Ed. 1991. *Systems Programming with Modula-3*. Prentice Hall.

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