Persistent Object Systems: Principles and Practice, Richard Connor and Scott Nettles (Eds.), Morgan Kaufmann, 1997

Residency check elimination for object-oriented persistent languages

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

Abstract

We explore the ramifications of object residency assumptions and their impact on residency checking for several subroutine dispatch scenarios: procedural, static object-oriented, and dynamic (virtual) object-oriented. We obtain dynamic counts of the residency checks necessary for execution of several benchmark persistent programs under each of these scenarios. The results reveal that significant reductions in the number of residency checks can be achieved through application of residency rules derived from the dispatch scenario under which a program executes, as well as additional constraints specific to the language in which it is implemented.
Keywords: residency checks, optimization, object-orientation, static/dynamic dispatch

1 Introduction

Persistent programming languages view permanent storage as a stable extension of volatile memory, in which objects may be dynamically allocated, but which persists from one invocation to the next. A persistent programming language and object store together preserve *object identity*: every object has a unique identifier (in essence an address, possibly abstract, in the store), objects can refer to other objects, forming graph structures, and they can be modified, with such modifications being visible in future accesses using the same unique object identifier. Access to persistent objects is *transparent* (at least from the programmer's perspective), without requiring explicit calls to read and write them. Rather, the language implementation and 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 [ABC$^+$83] has important ramifications for the design of persistent programming languages, since it encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax and store semantics. The notion of persistent storage as a stable extension of the dynamic allocation heap allows a uniform and transparent treatment of both transient and persistent data, with persistence being orthogonal to the way in which data is defined, allocated, and manipulated. This characterization of persistence allows us to identify the fundamental mechanisms that any transparent persistent system must support. Notable among these is the need for some kind of *residency check* to trigger retrieval of non-resident objects.

To be widely accepted, orthogonal persistence must exhibit sufficiently good performance to justify its inclusion as an important feature of any good programming language. We offer evidence that orthogonal persistence can be added to an object-oriented language without compromising performance. Our focus is on avoiding residency checks on objects when their residency can be guaranteed by the context in which their references are used. We consider several scenarios under which residency checks can be eliminated, and characterize the execution of a suite of benchmark persistent programs for each scenario in terms of the number of residency checks incurred by the benchmark. The scenarios represent a spectrum of styles of execution: procedural (i.e., non-object-oriented); object-oriented with static binding of methods to call sites; and object-oriented with dynamic method dispatch.

The remainder of the paper is organized as follows. We begin by reviewing object faulting and residency checking, followed by a description of the execution scenarios we consider. A discussion of the experimental framework follows, including description of the prototype persistent Smalltalk implementation used for the experiments, the benchmark programs and metrics used for evaluation, and presentation of results. Finally, we offer brief conclusions.

[*]See also: http://www.cs.purdue.edu/people/hosking. This work was supported by Sun Microsystems, Inc.

2 Object faulting and residency checking

As in traditional database systems, a persistent system caches frequently-accessed data in memory for efficient manipulation. Because (even virtual) memory may be a relatively scarce resource, it is reasonable to suppose that there will be much more persistent data than can be cached at once. Thus, the persistent system must arrange to make resident just those objects needed by the program for execution. Without knowing in advance which data is needed, the system must load objects on demand, from the persistent object store into memory. An *object fault* is an attempt to use a non-resident object. It relies on *residency checks,* which can be implemented explicitly in software, or performed implicitly in hardware and giving rise to some kind of hardware trap for non-resident objects. A wide range of object faulting schemes have been devised,[1] each having different representations for references to persistent objects. Some approaches drive all faulting with memory protection traps and make object faulting entirely transparent to compiled code; these have only one representation: virtual memory pointers to apparently resident objects. However, there is evidence to suggest that such totally transparent schemes do not always offer the best performance [HMS92, HM93a, HM93b, HBM93, Hos95, HM95]. Thus, multiple representations arise for references to resident objects (which can be used without causing an object fault), versus references to non-resident objects, along with explicit residency checks to distinguish them.

Efficient implementation of residency checks is one key to implementing a high-performance persistent programming language. The mechanism must be sufficiently lightweight as to represent only marginal overhead to frequently-executed operations on fine-grained objects. Nevertheless, even marginal overhead will have a cumulatively significant impact on overall performance. Thus, any opportunity should be exploited to elide residency checks where they are not strictly necessary [HM90, HM91, MH94]. Such optimizations rely on data flow analysis and code transformations (e.g., hoisting or combining residency checks) and the imposition of special rules about the residency of particular objects. Example rules and their ramifications include:

Pinning: *Objects once resident are guaranteed to remain resident so long as they are directly referenced from the machine registers and activation stacks (i.e., local variables).*

Thus, repeated residency checks on the same object referenced by a local variable can be merged into one check the first time the object is accessed through the variable.

Target residency: *The first argument of an object-oriented method call (i.e., the target object) will (somehow) automatically be made resident at the time of the call and remain so throughout.*

Thus, methods need not contain checks on the residency of their target object.

Coresidency: *Whenever object a is resident so also must object b be resident. This constraint is written $a \rightarrow b$.*

Thus, if a contains a reference to b, then b can be accessed directly from a (i.e., the reference from a to b can be traversed) without a residency check. Since a is resident as the source of the reference to b the coresidency constraint means that b will also be resident. For swizzling purposes, the reference from a to b is always represented as a direct memory pointer. Note that coresidency is asymmetric: $a \rightarrow b \not\Rightarrow b \rightarrow a$.

Pinning can be assumed to apply in all situations, since it enables all other residency check optimizations – in its absence no local variable can be guaranteed to refer to a resident object despite prior residency checks on that reference. The effect of the *target residency* and *coresidency* rules on the number of residency checks executed by a program is the topic of this paper. We consider several rule scenarios and measure the number of residency checks required under each scenario for execution of a suite of object-oriented persistent benchmark programs.

3 Execution scenarios

The residency rules to be applied at run-time dictate statically where residency checks are needed and where they can be elided. Our experiments include results for the following general execution scenarios:

Procedural: Execution in a non-object-oriented procedural language proceeds through the invocation of statically determined procedures. Ignoring possibilities for optimization of residency checks based on local/global data flow analysis, every dereference requires a residency check.

[1][ACC82, BC86, KK83, Kae86, CM84, RMS88, SMR89, BBB$^+$88, Ric89, RC90, Ric90, SCD90, WD92, HMB90, Hos91, HM93a, LLOW91, SKW92, WK92]

Static OO: Object-oriented programs execute through the invocation of methods on objects. A method typically accesses the encapsulated state of its target object. Thus, applying the *pinning* and *target residency* rules eliminates all residency checks on the target object of a method. Instead, a residency check on the target must be performed at the time the method is invoked, unless the method invocation is directed at the caller's own target object, in which case no check is needed. For non-virtual (i.e., statically bound) methods the method code is invoked directly so the target residency check must be performed explicitly prior to the call.

Dynamic OO: A defining feature of object-oriented languages is their support for inclusion polymorphism through mechanisms such as subclassing, subtyping and inheritance. Such polymorphism means that a given call site may involve target objects of any number of different but compatible types/classes. For virtual (i.e., dynamically dispatched) methods, the particular method code to be invoked is determined dynamically based on the type/class of the target object. Once again, we assume both *pinning* and *target residency*, but it is now possible to fold the target residency check into the dynamic method dispatch mechanism. The precise approach depends on the nature of the mechanism, but in general there is no additional overhead due to residency checking. Rather, the inherent indirection of dynamic dispatch is subverted, so that method invocations on non-resident objects are directed first to proxy faulting routines that make the target object resident, before forwarding the call to the appropriate resident object method. Again, no target object residency checks are necessary in the called method.

Note that although optimizations [Cha92, HU94, CG94, Fer95, DGC95, GDGC95, DMM96] may convert many indirect calls to direct calls, so increasing the number of explicit checks required, it is also likely that similarly aggressive optimizations can discover and eliminate redundant residency checks through intra- and inter-procedural data flow analysis.

In addition to these general scenarios regarding target object residency, a given program may benefit from *coresidency* rules that allow further elimination of residency checks. Such rules depend on the particular execution patterns of a given program. We consider the effect of specific coresidency rules below in the context of the prototype persistent system used in the experiments.

4 Experiments

We have instrumented the execution of several benchmark persistent programs executing in our prototype persistent Smalltalk system [Hos95] to obtain dynamic counts of residency checks performed under each of the above scenarios. We also consider the effect of additional coresidency constraints arising from specific knowledge of Smalltalk's bytecode instruction set and execution semantics.

4.1 A prototype implementation: Persistent Smalltalk

The prototype is an implementation of Smalltalk [GR83], extended for persistence. It has two components: a *virtual machine* and a *virtual image*.

The virtual machine implements the bytecode instruction set to which Smalltalk source code is compiled, along with certain *primitive methods* whose functionality is built directly into the virtual machine. These typically provide low-level access to the underlying hardware and operating system on which the virtual machine is implemented. For example, low-level floating point and integer arithmetic, indexed access to the fields of array objects, and object allocation, are all supported as primitives. A primitive method is invoked in exactly the same way as an ordinary method expressed as a sequence of Smalltalk expressions, but its implementation is not a compiled method. Rather, the virtual machine performs the primitive directly, without the need for a separate Smalltalk activation record. Since the primitives are coded by hand in the virtual machine, we are also able to hand-optimize the primitives to remove redundant checks. The compiler in a compiled persistent language might discover the same optimizations automatically through intra-procedural data flow analysis.

The virtual image is derived from Xerox PARC's Smalltalk-80 image, version 2.1, with minor modifications. It implements (in Smalltalk) all the functionality of a Smalltalk development environment, including editors, browsers, a debugger, the bytecode compiler, class libraries, etc. – all are first-class objects in the Smalltalk sense. Bootstrapping a (non-persistent) Smalltalk environment entails loading the entire virtual image into memory for execution by the virtual machine.

The persistent implementation of Smalltalk places the virtual image in the persistent store, and the environment is bootstrapped by loading just that subset of the objects in the image sufficient for resumption of execution by the virtual machine.

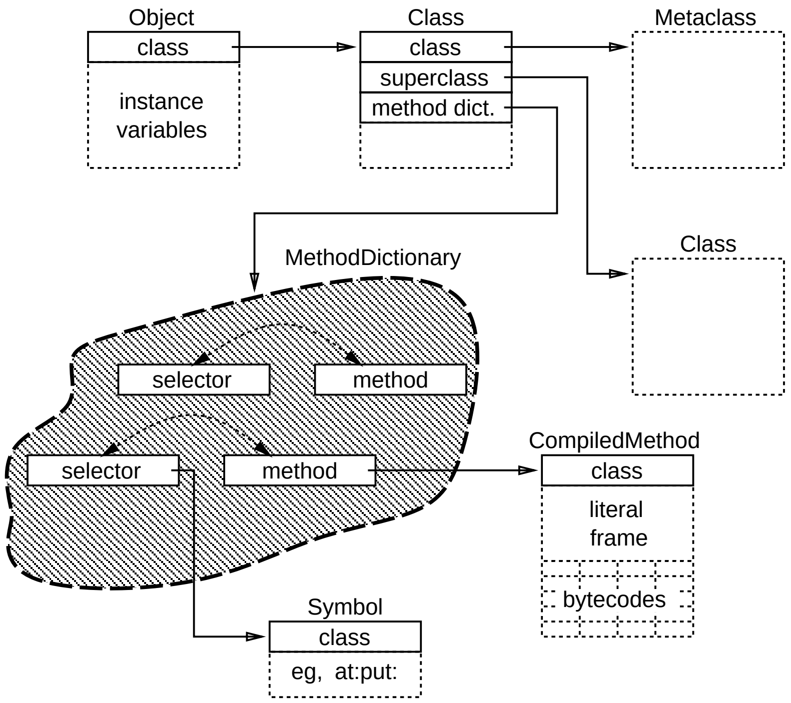

Figure 1: Objects, classes, inheritance, and method dispatch

We retain the original bytecode instruction set and make only minor modifications to the virtual image. Rather, our efforts focus on the virtual machine, which is carefully augmented with residency checks to fault objects into memory as they are needed by the executing image. The precise location of residency checks depends on the particular execution scenario.

4.1.1 Smalltalk method dispatch

A Smalltalk *object* (see Figure 1) is an encapsulation of some private state and a set of operations called its *interface*. The private state consists of a number of data fields, called *instance variables,* directly accessible only from the code implementing the object's operations. Every object is an *instance* of some *class* object, which implements the common behavior of all its instances; a class object is itself an instance of its *metaclass*. Classes are arranged in a hierarchy, such that a *subclass* will *inherit* instance behavior from its *superclass*. Thus, an instance of the subclass will behave as an instance of the superclass, except where the subclass overrides or extends that behavior.

Computation in Smalltalk proceeds through the sending of *messages* to objects. A message consists of a *message selector* (e.g., at:put:) and a number of arguments, and represents a request to an object to carry out one of its operations. The effect of sending a message is to invoke one of the *methods* of the object receiving the message (the *receiver*). Invoking a method may be thought of as a procedure call, with the receiver being the first argument to the procedure, preceding the arguments specified in the message. The particular method to execute is determined dynamically, using the message selector and the class of the receiver. Each class object contains a reference to a *method dictionary,* associating message selectors with *compiled methods*. A compiled method consists of the virtual machine bytecode instructions that implement the method, along with a *literal frame,* containing the shared variables,[2] constants, and message selectors referred to by the method's bytecodes.

Determining which method to execute in response to the sending of a message proceeds as follows. If the method dictionary of the receiver's class contains the message selector, then its associated method is invoked. Otherwise, the search continues in the superclass of the object, and so on, up the class hierarchy. If there is no matching selector in any of the method dictionaries in the hierarchy then a run-time error occurs.

[2]A shared variable is an object that encapsulates a reference to another object. If the contents of the variable are changed, then the change is visible to all other compiled methods holding references to that shared variable.

As described so far, the method lookup process is very expensive, especially since a given message may be implemented by a method in a class that is high up in the superclass hierarchy, far removed from the class of the receiver. A *method lookup cache* reduces this lookup cost significantly. A valid entry in the cache contains object references for a selector, a class, and a compiled method. Message sends first consult the method lookup cache, by hashing the object references of the selector and the receiver's class to index an entry in the cache. If the selector and class of the cache entry match those of the message, then the cached compiled method is invoked directly. Otherwise, the full method lookup locates the compiled method and loads the cache entry with the selector, class and method, before invoking the method.

4.1.2 The bytecode instruction set

We retain the standard Smalltalk-80 bytecode instruction set [GR83], which is partitioned by functionality as follows:

Stack bytecodes move object references between the evaluation stack of the current activation and:

1. the named instance variables of the receiver for that activation
2. the temporary variables local to that activation
3. the shared variables of the literal frame of the active method

Jump bytecodes change the instruction pointer of the current activation

Send bytecodes invoke compiled or primitive methods

Return bytecodes terminate execution of the current activation, and return control to the calling activation

4.2 Benchmarks

The performance evaluation draws on the OO1 object operations benchmarks [CS92] to compare the alternative execution scenarios. The operations are retrieval-oriented and operate on substantial data structures, although the benchmarks themselves are simple, and so easily understood. Their execution patterns include phases of intensive computation so that memory residence is important.

4.2.1 Benchmark database

The OO1 benchmark database consists of a collection of 20,000 *part* objects, indexed by part numbers in the range 1 through 20,000, with exactly three *connections* from each part to other parts. The connections are randomly selected to produce some locality of reference: 90% of the connections are to the "closest" 1% of parts, with the remainder being made to any randomly chosen part. Closeness is defined as parts with the numerically closest part numbers. We implement the part database and the benchmarks entirely in Smalltalk, including the B-tree used to index the parts.

The part objects are 68 bytes in size (including the object header). The three outgoing connections are stored directly in the part objects. The string fields associated with each part and connection are represented by references to separate Smalltalk objects of 24 bytes each. Similarly, a part's incoming connections are represented as a separate Smalltalk Array object containing references to the parts that are the source of each incoming connection. The B-tree index for the 20,000 parts consumes around 165KB.

4.2.2 Benchmark operations

The OO1 benchmarks comprise three separate operations:

Lookup fetches 1,000 randomly chosen parts from the database. A null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part (to ensure the part is actually made resident).

Traversal fetches all parts connected to a randomly chosen part, or to any part connected to it, up to seven hops (for a total of 3,280 parts, with possible duplicates). Similar to the Lookup benchmark, a null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part. OO1 also specifies a *reverse* Traversal operation, Reverse, which swaps “from” and “to” directions. Reverse is of minimal practical use because the random nature of connections means that the number of “from” connections varies among the parts – while every part has three *outgoing* connections, the number of *incoming* connections varies randomly. Thus, different iterations of the Reverse vary randomly in the number of objects they traverse, and so the amount of work they perform.

Insert allocates 100 new parts in the database, each with three connections to randomly selected parts as described in Section 4.2.1 (i.e., applying the same rules for locality of reference). The index structure must be updated, and the entire set of changes committed to disk.

Although this operation is a reasonable measure of update overhead, it is hampered by a lack of control over the number and distribution of the locations modified, and its mixing of updates to parts and the index. A more easily controlled benchmark is the following:

Update [WD92] operates in the same way as the Traversal measure, but instead of calling a null procedure it performs a simple update to each part object encountered, with some fixed probability. The update consists of incrementing the x and y scalar integer fields of the part. All changes must be reflected back to the persistent store. Here, the probability of update can vary from one run to the next to change the frequency and density of updates.

These benchmarks are intended to be representative of the data operations in many engineering applications. The Lookup benchmark emphasizes selective retrieval of objects based on their attributes, while the Traversal benchmark illuminates the cost of raw pointer traversal. The Update variant measures the costs of modifying objects and making those changes permanent. Additionally, the Insert benchmark measures both update overhead and the cost of creating new persistent objects.

4.3 Metrics

We obtain dynamic counts of the number of residency checks necessary for the execution of the benchmark operations using an instrumented version of the Smalltalk virtual machine. A benchmark *run* consists of ten iterations of the benchmark operation. Because each successive iteration accesses a *different* set of random parts, we characterize each benchmark in terms of the mean number of residency checks for the 10 iterations of the run, and calculate 90% confidence intervals to bound the variation among random iterations. Using different counters for each possible scenario enables the results for all scenarios to be gathered with just one run of each benchmark. Thus, each scenario sees the same run of random iterations.

4.4 Results

The initial statement of results ignores residency check counts attributable to Smalltalk’s idiosyncratic treatment of classes, activation records, compiled methods, and process stacks as (orthogonally persistent) objects in their own right. Thus, the counts do not reflect residency checks needed when ascending the class hierarchy during method lookup for dynamic method dispatch, nor residency checks on processes and stacks during process management, and checks on activation records during returns. We do this so as to obtain the closest possible analogy to more traditional languages such as C, C^{++} and Modula-3, in which dynamic method dispatch is implemented as an indirect call through a method table associated with the target object, and which do not treat processes, activations, and classes/types as first-class objects. The intricacies of residency checks for such complications are discussed later.

The results appear in Table 1, with columns for each of the execution scenarios, and rows for each benchmark. The number of residency checks required for execution of the benchmark under each execution scenario appears along with the fraction of checks that can be elided in light of the scenario’s residency rules. We also indicate the percentage of method invocations that result in primitive method executions. Recall that primitive methods are hand-optimized to minimize the number of residency checks necessary for their execution based on the access patterns of the primitive. Also, only primitives can directly access objects other than the target object; non-primitives must instead invoke a method on non-target objects. Thus, *target residency* optimizations are likely to be more effective when the ratio of primitives to non-primitives is low, since fewer non-target accesses will occur.

It is clear that the *target residency* rule significantly reduces the number of checks necessary under the object-oriented execution scenarios. The statically dispatched scenario, for which method invocations on objects other than the caller’s target require a check, is able to eliminate 24–75% of checks, depending on the benchmark. The remaining checks are necessary

Benchmark	Execution scenario						Primitives versus non-primitives
	Procedural		Static OO		Dynamic OO		
	Checks	elided	Checks	elided	Checks	elided	
Lookup	44661± 29	0%	22330± 15	50%	1002± 0	97%	83%
Traversal	13158± 0	0%	3275± 8	75%	1± 0	99%	0%
Reverse	28106±8238	0%	13234±3884	52%	5880±1725	79%	33%
Update							
0%	12855± 0	0%	9738± 8	24%	1694± 0	86%	56%
5%	13481± 77	0%	9738± 8	27%	1694± 0	87%	56%
10%	14104± 101	0%	9738± 8	30%	1694± 0	87%	56%
15%	14753± 114	0%	9738± 8	33%	1694± 0	88%	56%
20%	15437± 110	0%	9738± 8	36%	1694± 0	89%	56%
50%	19311± 96	0%	9738± 8	49%	1694± 0	91%	56%
100%	25975± 0	0%	9738± 8	52%	1694± 0	93%	56%
Insert	30557± 423	0%	20026± 393	34%	2203± 122	92%	82%

(interval confidence is 90%)

Table 1: Residency checks by execution scenario and benchmark

because of invocations on objects other than the caller's target, and primitive accesses to objects other than the primitive callee's target.

The dynamic scenario eliminates the need for all checks on method invocation since target residency checking is folded into the indirect, dynamic method dispatch. As a result, this scenario requires 86–99% fewer residency checks than for procedural execution. The remaining checks are necessary as a result of primitive access to objects other than the target. In fact, it turns out that for these benchmarks the remaining checks are solely on arguments to primitives. The variation in the ratio of primitive to non-primitive checks illustrates this directly – where the primitive fraction is low (as in Traversal), a higher fraction of the checks are elided.[3]

4.5 Smalltalk complications

As mentioned earlier, there are additional complications for a persistent Smalltalk implementation, arising out of Smalltalk's treatment of control objects such as processes, activation stacks, and classes as first-class objects that can themselves persist. We add *coresidency* rules to eliminate checks on these objects as follows:

Class coresidency: An object's class is always coresident with each of its instances. Thus, the send bytecodes need not perform a residency check on the target object's class when probing the method lookup cache.

Sender coresidency: For any stack frame object, the stack frame representing its sender (i.e., calling) activation is always coresident. Applying this rule transitively results in all activations in a process stack being coresident – when an active stack frame is made resident (usually because its process is being resumed), its caller, its caller's caller, and so on up the process stack, are made resident along with it. Since the return bytecodes directly manipulate the active stack frame and the (calling) activation to which control is being returned, sender coresidency eliminates the need for a residency check on the caller in the return bytecodes.

Method coresidency: Methods are always coresident with their activation's stack frame, since an activation can only execute if its corresponding compiled method is resident. Thus, return bytecodes need not check the residency of the method in which execution resumes.

Literal coresidency: Literals are always coresident with the methods that refer to them. They include the selectors, constants and shared variables directly manipulated by the bytecodes of the method. Send bytecodes directly access literal selectors and certain stack bytecodes directly access shared variables. Thus, these bytecodes need not check the residency of the literals they manipulate.

[3]Multi-methods, as in Cecil [Cha95], in which method dispatch occurs on more than one argument of the method, would submit to folding of residency checks on all qualified arguments into the indirection of dispatch.

These special coresidency rules for Smalltalk force preloading of objects critical to the forward progress of computation, so that *all bytecode instructions of the persistent virtual machine execute without residency checks*. The persistent virtual machine must still check the residency of objects whose residency is not guaranteed by these rules. For example, full method lookup requires checks as it ascends the class hierarchy, to ensure that the superclasses and their method dictionaries are resident. Similarly, primitive methods must perform residency checks on objects they access directly (excluding the receiver, guaranteed resident by the *target residency* rule).

5 Conclusions

We examined the impact of several execution scenarios on the residency checks necessary for execution of several instrumented benchmark programs. The results indicate that the object-oriented execution paradigm enables a significant reduction in residency checks through the simple application of the target object residency rule. In addition, coresidency constraints specific to the persistent Smalltalk prototype allow a further reduction in the number of checks required, so that the bytecode instructions of the persistent Smalltalk virtual machine are able to execute without any residency checks at all. It would be interesting to consider the application of similar techniques for persistence to other dynamic object-oriented languages, such as Java [GJS96, LY96].

A particularly promising avenue of further research is how optimization can both hinder (e.g., through aggressive elimination of dynamic method dispatch) and promote (e.g., through exploitation of coresidency rules specific to the application program, as well as discovery of residency invariants through data flow analysis) the elimination of residency checks.

References

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[ACC82] Malcolm Atkinson, Ken Chisolm, and Paul Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[BBB+88] Francois Bancilhon, Gilles Barbedette, Véronique Benzaken, Claude Delobel, Sophie Gamerman, Christophe Lécluse, Patrick Pfeffer, Philippe Richard, and Fernando Velez. The design and implementation of O_2, an object-oriented database system. In Dittrich [Dit88], pages 1–22.

[BC86] A. L. Brown and W. P. Cockshott. The CPOMS persistent object management system. Technical Report PPRP 13, University of St. Andrews, Scotland, 1986.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994.

[Cha92] Craig Chambers. *The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages*. PhD thesis, Stanford University, 1992.

[Cha95] Craig Chambers. The CECIL language: specification and rationale. Version 2.0. http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html, December 1995.

[CM84] George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of the ACM International Conference on Management of Data*, pages 316–325, Boston, Massachusetts, June 1984.

[CS92] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, March 1992.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[Dit88] K. R. Dittrich, editor. *Proceedings of the International Workshop on Object Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, Bad Münster am Stein-Ebernburg, Germany, September 1988. *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. 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, October 1996.

[DSZ90] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Proceedings of the International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, September 1990. *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990.

[Fer95] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 103–115, La Jolla, California, June 1995.

[GDGC95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Austin, Texas, October 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HBM93] Antony L. Hosking, Eric Brown, and J. Eliot B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the International Conference on Very Large Data Bases*, pages 429–440, Dublin, Ireland, August 1993. Morgan Kaufmann.

[HM90] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimisations for persistence. In Dearle et al. [DSZ90], pages 17–27.

[HM91] Antony L. Hosking and J. Eliot B. Moss. Compiler support for persistent programming. Technical Report 91-25, Department of Computer Science, University of Massachusetts at Amherst, March 1991.

[HM93a] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 288–303, Washington, DC, October 1993.

[HM93b] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 106–119, Asheville, North Carolina, December 1993.

[HM95] Antony L. Hosking and J. Eliot B. Moss. Lightweight write detection and checkpointing for fine-grained persistence. Technical Report 95-084, Department of Computer Sciences, Purdue University, December 1995.

[HMB90] Antony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an object faulting persistent Smalltalk. Technical Report 90-45, Department of Computer Science, University of Massachusetts at Amherst, May 1990.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992.

[Hos91] Antony L. Hosking. Main memory management for persistence, October 1991. Position paper presented at the OOPSLA'91 Workshop on Garbage Collection.

[Hos95] Antony L. Hosking. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. PhD thesis, University of Massachusetts at Amherst, February 1995. Available as Computer Science Technical Report 95-02.

[HU94] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 326–336, Orlando, Florida, June 1994.

[Kae86] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 87–106, Portland, Oregon, September 1986.

[KK83] Ted Kaehler and Glenn Krasner. LOOM—large object-oriented memory for Smalltalk-80 systems. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 14, pages 251–270. Addison-Wesley, 1983.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[MH94] J. Eliot B. Moss and Antony L. Hosking. Expressing object residency optimizations using pointer type annotations. In Malcolm Atkinson, David Maier, and Véronique Benzaken, editors, *Proceedings of the International Workshop on Persistent Object Systems*, pages 3–15, Tarascon, France, September 1994. Springer-Verlag, 1995.

[RC90] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software: Practice and Experience*, 19(12):1115–1150, December 1990.

[Ric89] Joel Edward Richardson. *E: A Persistent Systems Implementation Language*. PhD thesis, University of Wisconsin – Madison, August 1989. Available as Computer Sciences Technical Report 868.

[Ric90] Joel E. Richardson. Compiled item faulting: A new technique for managing I/O in a persistent language. In Dearle et al. [DSZ90], pages 3–16.

[RMS88] Steve Riegel, Fred Mellender, and Andrew Straw. Integration of database management with an object-oriented programming language. In Dittrich [Dit88], pages 317–322.

[SCD90] D. Schuh, M. Carey, and D. DeWitt. Persistence in E revisited—implementation experiences. In Dearle et al. [DSZ90], pages 345–359.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas, an efficient, portable persistent store. In Antonio Albano and Ronald Morrison, editors, *Proceedings of the International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag, 1992.

[SMR89] Andrew Straw, Fred Mellender, and Steve Riegel. Object management in a persistent Smalltalk system. *Software: Practice and Experience*, 19(8):719–737, August 1989.

[WD92] Seth J. White and David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the International Conference on Very Large Data Bases*, pages 419–431, Vancouver, Canada, August 1992. Morgan Kaufmann.

[WK92] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE.