Expressing Object Residency Optimizations Using Pointer Type Annotations

J. Eliot B. Moss and Antony L. Hosking
Object Systems Laboratory, Department of Computer Science
University of Massachusetts; Amherst, MA 01003, USA
Sixth International Workshop on Persistent Object Systems
Tarascon, Provence, France, 5–9 September 1994

Abstract

We consider some issues in optimizing persistent programming languages. In particular, we show how to express optimizations of object residency checks in strongly typed persistent languages as "annotations" on pointer types. These annotations essentially extend and refine the type system of the language, and they have at least two significant uses. First, a programmer can use them to express desired residency properties to be enforced by the language implementation (compiler plus run time). Second, we can use them to separate a persistence optimizer, which adds annotations, from the remainder of the compiler, which simply obeys them. This gives rise to a nice separation of concerns in supporting high-performance persistence: the "intelligent" optimizer can be factored off from the rest of the compiler.

In addition to modularity benefits, the separation allows us to explore the value of various optimizations without actually implementing them in the optimizer. Rather, we can optimize programs by hand and compare optimized and unoptimized code to develop sound data to use when deciding whether to implement an optimization. While the approach is similar to source-to-source optimizers, which are by no means a new idea, in our case the target language is an extension of the source language, and one specifically designed to be *easier* to compile well. We are applying the approach in our ongoing implementation of Persistent Modula-3. We present the type annotation approach in the context of Modula-3, but it should be applicable to any strongly typed persistent programming language, as well as to a range of other kinds of optimizations.

1 Introduction and Motivation

We have been developing run time support and an optimizing compiler for Persistent Modula-3[1] for some time. In the process we have conceived of a number of optimizations one might consider to improve performance of persistent programs [HM90, HM91], and have compared several approaches to dealing with (among other things) *object faults* (attempts to use persistent objects that are not currently resident (i.e., not actually in the process's virtual address space)), in our Persistent Smalltalk implementation [HM93a, HM93b]. Most of the optimizations we have thought of require reasonably powerful data flow analysis and code transformations, such as hoisting or combining residency checks, or imposing special rules that complicate the compiler such as: "the first argument of

[1] For further information on Modula-3 see [CDG+88, CDG+89, CDG+91, Nel91, Har92]; for further information on our persistence work see [MS88, Mos89, Mos87, Mos90, HMB90, HM90, HM91, Hos91, Mos92, HM93a, HBM93, HM93b].

a method call (i.e., the target object) will (somehow) automatically be made resident throughout the execution of the method" (so that the method code need not contain checks on uses of the target object). Implementing these optimizations would require a lot of effort in the back end of a modern optimizing compiler (such as the GNU C compiler, whose back end we use in this work). We concluded that it would be best to explore the effectiveness of a variety of optimizations before trying to implement them.

We were willing to hand optimize a collection of benchmark programs, but we still needed a way to express the possible *results* of the optimizations. The point here is to be able to control the code emitted by the compiler. Since we were working in the context of a statically typed, compiled, object-oriented language (Modula-3 with persistence extensions), we decided to try expressing the results of optimizations in the type system, giving rise to the overall approach that is the point of this paper.

We organize the remainder of the presentation as follows. First we describe relevant aspects of Persistent (and non-persistent) Modula-3. Then we briefly review object faulting and object residency checking, and enumerate some ways of implementing them. Next, we argue for residency check optimization and present a list of residency related optimizations one might want to explore. We then explain how we use types and type conversions to express residency knowledge and residency checks, and show how the approach can express the desired optimizations in the various implementation approaches. We finish with a few concluding remarks.

2 Relevant Aspects of Persistent Modula-3

Modula-3 is a strongly typed object-oriented language in the tradition of Pascal and Modula-2. To Modula-2 it adds: object types (in a single inheritance hierarchy), automatic storage management (garbage collection), exception handling, threads (lightweight processes running in the same address space), and generic interfaces and modules (templates that are expanded syntactically as needed, to form members of a parameterized family of interfaces and modules). Here we are most concerned with types, procedure calls, and method calls in Modula-3, and assume the reader can grasp other constructs intuitively from a knowledge of Pascal or Modula-2.

Unlike some object oriented languages (e.g., Smalltalk [GR83] and Trellis/Owl [SCB+86]), Modula-3 is not *uniformly* object oriented: it has a full collection of non-object primitive types (`INTEGER`, `REAL`, range, enumeration, etc.) and type constructors (`RECORD`, `ARRAY`, `REF`, `PROC`, etc.) in addition to object types. Again we assume that our example code fragments can be grasped intuitively from knowledge of Pascal. A Modula-3 object type consists of a *supertype*, which must be another object type (possibly the built in type `ROOT`), zero or more (new) *fields*, declared and used analogously to record fields, zero or more (new) *methods*, which are names bound to procedures to be invoked when the named method is called, and zero or more *overrides* of supertype method bindings, with the following syntax:

[*super*] `OBJECT` *fields* [`METHODS` *methods*] [`OVERRIDES` *overrides*] `END`

This example shows a trivial `point` data type, giving only the methods for manipulating points, without implementations, and then giving representation and implementation details in a subtype.[2]

[2] In Modula-3 one would actually use *opaque types* to hide the representation type `p_rep` and yet have it be the actual type used for `point`. The type `point` here has no data and no method implementations; it is useful

```
TYPE
  point = ROOT OBJECT
              METHODS scale(by: REAL): point; ...
              END;
  p_rep  = point OBJECT x, y: INTEGER;
               OVERRIDES scale := scale_point;
               END;
```

Modula-3 allows (appropriately constrained) self-reference, and use of names in a scope without respect to their order of declaration, so it is easy to define recursive types, such as this:

```
TYPE ilist = REF RECORD i: INTEGER; next: ilist; END;
```

It is worthwhile to note that Modula-3 object types are implicitly pointers to dynamically allocated memory containing the object fields (and a reference to the method suite of the object[3]), thus a more object oriented form of the integer list type would be:

```
TYPE olist = OBJECT i: INTEGER; next: olist; END;
```

Modula-3 procedure call is straightforward. There are three argument binding modes: by value, which is the default, by reference, indicated by `VAR`, and by reference but without rights to modify the argument, indicated by `READONLY`. Modula-3 exception handling is irrelevant here, so we will not describe it. A Modula-3 method call is in most respects like an ordinary procedure call. However, the target object is not explicitly listed in the method's call interface (being implied from the object type), as can be seen from `scale` in the `point` object type example. Note, though, that the procedure bound to `scale` *would* include the target object as its first argument, for example:

```
PROCEDURE scale_point (p: p_rep; by: REAL): p_rep = ...
```

The language's (sub)type checking rules allow `scale_point` as an implementation of `scale` in this case. Here is an example of a method call:

```
p: point := ...;
p.scale(2.5);
```

Thus far we have described non-persistent Modula-3. To add persistence to Modula-3, we changed very little [HM90]: we re-interpreted `REF t` to mean a reference to a (possibly) persistent instance of `t`, and likewise for object types. By "possibly persistent" we mean that newly created instances need not be created in the store (as opposed to existing only in main memory), unless they are reachable from a persistent root object at a designated time (checkpoint). As a side note, we observe that Modula-3 includes `UNTRACED REF t` in addition to `REF t`; untraced pointer types are managed with explicit allocation and deallocation and are not traced by the garbage collector. They are useful on occasion, e.g., for allocating fixed I/O buffers and the like. In [HM90] we added `TRANSIENT REF t`. By analogy with untraced types, transient pointer types refer to instances that can *never* become persistent. Again, they appear to have occasional uses, but from here on we will ignore untraced and transient pointer types since they are irrelevant here.

---

only to document the interface and as a foundation for implementations. Since opaque types are not relevant to this work, we discuss them no further.

[3] A Modula-3 method suite is a vector of pointers to code for methods; the C++ terminology is "virtual function table".

3 Object Faulting and Residency Checking

The whole idea of a persistent programming language is to provide transparent access to objects maintained in a persistent object store. Unlike simply reading and writing blocks of data using traditional file I/O, 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.

Given adequate address space, one could read in (or map into virtual memory) the entire object store, but, as many others do (and for good reasons we will not get into here), we assume that such an implementation strategy is not preferred. Thus, as a program runs, we will need to load objects on demand, from the persistent object store into main (or virtual) memory. An *object fault* is an attempt to use a non-resident object. Object faulting relies on *object 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. Object faulting also relies on mechanisms to load objects from the store into memory, and ultimately to write (at least the modified) objects back. In a complete environment one must also integrate support for concurrency control, recovery, and distribution, but we focus primarily on object residency aspects of persistent systems here.

A wide range of object faulting *schemes* have been devised ([ACC82, BC86, KK83, Kae86, CM84, RMS88, SMR89, BBB$^+$88, Ric89, SCD90, WD92, HMB90, Hos91, HM93a, LLOW91, SKW92, WK92] are not exhaustive). Any scheme has some number of distinct *situations*, such as a reference to a resident object (which presumably can be used without causing an object fault), versus a reference to a non-resident object. We are concerned only with situations that require the compiler to generate *distinct code*; such distinct situations give rise to corresponding *representations*. For example, in schemes that drive all faulting with memory protection traps and make object faulting entirely transparent to compiled code (such as [SMR89, SCD90, LLOW91, SKW92]), there is only one representation: apparently resident objects. However, we have gathered evidence that such totally transparent schemes do not always offer the best performance [HM93a, HM93b, HBM93, HMS92]. Hence we will be most interested in schemes that have more than one representation.

We list below a range of possible representations. Any given scheme may combine a number of them, though not all subsets make a lot of sense. Also, at any given point in a program, an optimizer can develop an upper bound on the representations possible for each pointer, but in general multiple representations may be possible (e.g., it may not be able to establish whether a given object is resident or not, and the two situations may have different representations). We further observe that schemes differ in whether, when, and how they *swizzle* pointers, i.e., convert between (possibly) different formats used in the persistent store and in main memory.[4]

Direct pointer: a direct (virtual memory) pointer to an apparently resident object; either the object is actually resident, or hardware traps may be used to make it resident if the pointer is used; requires no check[5]

[4] For more background on swizzling, the reader may start with [Mos92, WD92, WK92].

[5] While the pointer manipulation code is the same, pointers to resident objects (arranged by prefetching, etc.), and pointers that will trap when used, have different performance characteristics, and ultimately we might care to distinguish between them.

Object Identifier: a unique object identifier, which requires some kind of table lookup to locate the corresponding object's data in memory; this is assumed complex enough to require a call, and the lookup routine will make the object resident if it is not yet so; avoids swizzling but may incur repeated lookup costs

Indirect pointer: a pointer to a cell containing a pointer to the object's contents; similar to the direct pointer scheme, but this is more flexible in that one need not allocate address space for object contents prior to loading the object (and hence need not know the object's true size in advance); requires no check

Fault block: a pointer to a fixed size block of memory that contains the object's unique identifier; causes a fault when used

Proxy object: a pointer to an object that stands in for a non-resident object; has a method suite, all of whose methods will load the true object; field access must load the object, but method call is transparent

Indirect object: a pointer to an object that stands in for a (now) resident object; has a method suite, all of whose methods forward calls to the true object; field access must forward explicitly, but method call is transparent.

A number of schemes can be developed by choosing appropriate subsets of these representations. Of course, when representations require different code, they must be distinguishable one from another, so that when a pointer has multiple possible representations at a given use, the compiler can generate tests to discriminate. Here are a few schemes, to give a sense of the possibilities:

- Direct pointer (only): completely transparent, requires either hardware traps or preloading all reachable objects
- Direct pointer + fault block: used in our Persistent Smalltalk implementation, requires explicit checks to discriminate (which, by system design, we localized primarily to method invocation)
- Direct pointer + object identifier: requires a tag bit in pointers; may result in excess object identifier lookups, which are partly avoided by faulting on load of a pointer ("swizzle on discovery") rather than on use of it
- Object identifier (only): useful if a pointer is not traversed very many times during a program's execution (avoids overhead of swizzling)
- Direct pointer + proxy object + indirect object: proxy, indirect, and ordinary objects must be distinguishable on field access, but method call is transparent; field access can be turned into method call to gain complete transparency (but possibly higher overhead)
- Direct pointer + proxy object: avoiding indirect objects speeds use of resident objects, but requires removal of indirections when objects are loaded, which can be costly

4 Some Residency Optimizations

We now consider residency checking optimizations one might want to use in trying to improve performance of programs written in a persistent programming language. Here is a list of some optimizations we have thought of in our work:

Local subsumption: If we dereference the same pointer multiple times in a piece of straight line code, the first use will force residency. If we additionally impose an appropriate rule *pinning* the object, i.e., preventing it from from being removed from the address space, at least until the last use in the straight line code, then remaining uses need not check residency. We note again that subsumption is similar to common subexpression elimination, but differs in that it has a side effect (but is idempotent).[6]

Global subsumption: Given appropriate data flow analysis, one can apply subsumption across basic blocks, intra- or inter-procedurally. The result is similar to *hoisting* common subexpressions.

Target object residency: Since invoking a method requires first making the target object resident, method code can assume target residency. This could yield great improvements in programs written in object-oriented style.[7]

Formal parameter assertions: It may be useful to require a procedure or method argument to be resident before making a call. If the caller knows the object is resident, no checks are needed in either the caller or the callee. If we only use hoisting, and perform residency checks near the beginning of a procedure, we cannot eliminate checks where the caller knows the object is resident. Formal parameter assertions are especially useful for "internal" methods (called only from public methods), some recursive calls, and non-object-oriented code.

Procedure result assertions: If a procedure returns a newly created object, or one guaranteed to be resident (either because the procedure caused it to be or because the object was guaranteed to be resident on entry to the procedure), then it can help the caller and other downstream code to know that.

Data type assertions: In the case of a data type with multiple levels of pointers, it might be convenient to fault in several levels of objects at once (they may arrive together anyway, with proper clustering[8]), and avoid checks when traversing the levels. This can be accomplished by associating residency assertions with pointers inside data structures. We observe that for *recursive* data structures, placing such assertions on the recursion pointers (such as the `next` field of our `ilist` type example) may require a large closure of objects to be made resident (but may be a good idea if the objects are likely to be used).

---

[6] The original implementation of the E programming language [Ric90, RC90, Ric89] included an optimization similar to subsumption. It operated in a more general model, where the unit of access was byte ranges of objects, and could unify overlapping ranges in both space and time. However, it was applied in a somewhat ad hoc manner in the *cfront* implementation of E, and abandoned in later versions of E (after Richardson's departure from the group).

[7] Actually, type inference and other techniques might enable the compiler to know the precise type and avoid an object oriented dispatch, in which case it might need to introduce an explicit check.

[8] Clustering is an important issue because it affects I/O performance, which can be more noticeable than incremental CPU overheads, but it is outside the scope of this paper. However, see the discussion of clustering towards the end of the paper.

A converse sort of assertion would be that a reference is rarely traversed, and would best be kept in object identifier form and looked up on each use.

Of these optimizations, subsumption is likely always to be profitable—its only negative effect is pinning objects longer (and requiring support for such pinning, which may in turn require compiler produced tables for the run time system to use in determining which objects are pinned (along the lines of compiler support for accurate garbage collection [DMH92, Diw91, HMDW91])). Similarly, target object residency is probably almost always a good idea: it would be rare for the target of a message not to be used *and* for the method to be statically known. Formal parameter assertions require more care to prevent objects from being loaded if they are not *always* used. Data type assertions run even more risks, but can have high payoff. They may require profile feedback in addition to static analysis for an optimizer to make good decisions.

5 Using Types in Residency Optimization

Now we describe in more detail the central idea of the paper, which is to express persistence representation assertions as qualifications on pointer types. It is convenient to choose a particular, fairly interesting, scheme as a basis for presentation; we trust the application to other schemes will then be obvious. Our example scheme's representations are proxy objects, direct pointers, and object identifiers. Now a scheme includes exactly the possible degrees of knowledge the implementation may have concerning object residency: each state of knowledge can be described as a non-empty subset of the set of allowed representations. In our scheme we allow precisely these subsets:

$$\{ \text{ proxy, direct } \}, \{ \text{ direct } \}, \{ \text{ object identifier } \}$$

Why these? First, these groupings avoid the need to discriminate dynamically between pointers and object identifiers, which puts fewer constraints on the representation of object identifiers. That explains the absence of any other subsets containing "object identifier". Second, $\{$proxy$\}$ just does not seem very useful, since it requires work in exactly the same cases as $\{$proxy, direct pointer$\}$. Note that one can certainly conceive of allowing more subsets, etc. Clearly there are many possible schemes in our approach!

We use `REF t` to indicate $\{$proxy, direct pointer$\}$ knowledge, `RES REF t` (for *resident reference*) for $\{$direct pointer$\}$, and `ID REF t` for $\{$object identifier$\}$; we use similar annotations on object types.[9] In some sense these types do not all support the same operations: an `ID REF` requires lookup to dereference it or perform a method call, a `REF` requires discrimination and possible conversion on dereference (but not method call), a `RES REF` supports all operations directly. We can define a strict language, where field access is permitted only via a `RES REF`, and method call is allowed on `REF` or `RES REF`, and we supply conversion operators between all pairs of annotations. Note that all these kinds of pointers are equivalent at the level of language semantics—they differ only in their implementation properties. The situation is analogous to packed and unpacked data structures, which represent the same values in different ways. The strict language might be tiresome for humans. It is easy to extend the strict language to allow all operations on all pointer representations, with the necessary conversions being implied, into temporary variables that are used and then discarded. We will posit built-in functions `TO_REF`, `TO_RES`, and `TO_ID` that perform explicit conversions (again the syntax is

[9] The syntax really does not matter much for our purposes. In practice one might use pragmas rather than new keywords, so that annotated programs can still be processed by unextended compilers, etc.

not that important here). Let us now consider how the various optimizations can be represented using these annotations.

Local subsumption: We can introduce a new local variable that is a `RES REF`, and use it multiple times:

Unoptimized	Optimized
	`tmp: RES REF t := TO_RES(p);`
`... pˆ.x ...`	`... tmpˆ.x ...`
`... pˆ.y ...`	`... tmpˆ.y ...`

Global subsumption: Again, we can introduce a new `RES REF` local variable, set it at the point where we want to hoist the residency check, and use it subsequently (no example needed).

Target object residency: The issue here is having a way to express, for each method of an object type, its target object residency assumptions, and to insure that the procedures bound to methods conform to those assumptions. One way to accomplish this is to associate annotations with method names in object types, e.g.,:

```
TYPE
  point = ROOT OBJECT
              METHODS RES scale(by: REAL): point; ...
              END;
```

Any procedure bound to `scale` must declare its first argument in a manner compatible with the method's `RES` annotation. In this case `RES REF` or plain `REF` would be all right, but `ID REF` would not be.

Formal parameter and result assertions: Similar to target object residency tags, we associate tags with procedure formals and results:

```
PROCEDURE scale_point (p: RES(p_rep); by: REAL):
                      RES(p_rep) = ...
```

This is more general than target object residency annotations because it can apply to any argument position for a procedure, and because it applies to ordinary procedure call in addition to method calls. Any necessary conversions are performed by the caller, either to arguments (before the call) or results (after the call, and only if the result is used). Note that here the `RES` built-in function is being applied to *types* rather than objects.

Data type assertion: It is easy to devise a form for these assertions: we allow residency annotations on the types used in record fields, object fields, arrays, etc., not unlike the parameter and result annotations. The meaning and the manipulation rules are a bit more subtle, however. Consider this recursive type declaration:

```
TYPE rolist = RES OBJECT i: INTEGER; next: rolist; END;
```

Here the entire list must be resident. If the entire list is not accessed, this may load more objects than necessary. (One can use hardware trap driven loading rather than software pre-loading to prevent this over-loading, but hardware traps come with their own fielding costs.) This only indicates that the assertions must be used with care. We have more of a problem if we wish to use different annotations with the same underlying type, in the same program. For example, suppose we have both the declaration above and this one:

```
TYPE idolist = ID OBJECT i: INTEGER; next: idolist; END;
```

At first glance one might think that we could convert between object references of type `rolist` and type `idolist`, though it might involve traversing the list and converting all the embedded references. Unfortunately, this does not work because the list could then be referred to via pointers of both types *simultaneously*, which gives rise to inconsistent assertions on the `next` field of the objects. This difficulty is not an artifact of our notation, but is inherent: a given program must represent a given object consistently.[10] Similar concerns precluded subtyping between record types (etc.) in Modula-3 (see the "How the Language Got Its Spots" discussions in [Nel91]). It is not yet clear whether the restriction to consistent annotations has significant performance impact.

As an additional note, we observe that one can "unroll" a recursive type any fixed number of times, and place different assertions at each place around the type "loop". However, since this appears to require similar unrolling of *all* related recursive procedures (as well as iterative loops), it is not obvious one would want to do it very often. (A similar optimization has been suggested for a Standard ML implementation [SRA94].)

6 Clustering

While we have focused on the problem of residency check optimization, the annotations we propose also bear some relation to clustering and prefetching of persistent data, especially in the case of data type assertions. In particular, if a data type annotation indicates that the target of a given pointer field should be resident, then we have two basic options in the implementation: force residency when the "root" of the data structure is made resident, or use a transparent (probably protection trapping) technique. Since the code generated is the same in each case, one can actually decide at run time what to do. For example, one can set up pointers for objects that are actually resident or that arrive in the same cluster as the root object, and use protection traps for excursions outside that region. A possibly interesting hybrid approach is to set protection traps while simultaneously requesting the non-resident clusters. When the clusters arrive, objects could be swizzled and page protections turned off (unless, of course the application has already hit the protection barrier, in which case it must wait anyway).

In any case, the annotations can be thought of either as expressing *existing* clustering (to the extent that such a static method can express it), or expressing *desired* clustering. Note, though, that clustering is a global decision affecting all applications using the same data, so if the different applications have different clustering preferences, we have to reach some compromise. (One might replicate data with different clustering, but this also induces tradeoffs if the data are updated often.)

The essential point is that while the clustering problem is *similar to* the residency check optimization problem, and while we might use similar annotation schemes for each, we probably need separate annotations for clustering and residency optimization. We also observe that while the annotations we have discussed are adequate to control the code produced, they are not adequate for more sophisticated control of prefetch. Prefetch is more closely related to residency, so we may prefer annotations that integrate residency and prefetch guidance to the compiler and run-time system, with clustering handled separately.

[10] Note, though, that residency annotations essentially do not exist in the object store, and different programs can use *different* annotations with no problems.

7 Conclusion

We have shown how one can control fairly precisely the placement of residency checks and conversions, and express a range of interesting residency optimizations, using pointer type annotations. The technique appears to be reasonably simple and should be effective in its goals of separating residency optimization decisions from their implementation. Of course the general idea is not new—source to source transformations are widely used in optimizing programs, especially for parallel machines. Even the type tagging approach was suggested by Modula-3's UNTRACED REF and REF types, and the notion of multiple representations of the same value is also fairly obvious (e.g., packed versus unpacked types). Similar annotation techniques have been used in distributed programming languages to distinguish local versus remote, and so forth [BHJ+87]. Perhaps we can take credit for a slightly new application of old ideas.

While we tentatively conclude that the approach is effective towards our goals, what are its limits? How else might the general technique be used? Since similar annotations have been used in distributed programming languages, the approach seems to extend to that situation. We also believe that pointer type annotations might be applied successfully in indicating compiler knowledge of concurrency control status. In a lock based system we might distinguish unlocked, share mode locked, and exclusive mode locked. Unlike the residency case, where we left unpinning an object as somewhat vague and unspecified, with concurrency control it is probably important to be clear about when an object becomes unlocked. The difference is that locking is semantically significant, whereas residency optimization affects only performance, not semantics. Another possible application of pointer type annotations is in reducing work in recording which objects are modified as a persistent program executes. Modification noting is analogous to residency checking in that it is idempotent (provided the object is not written back to the store in between the modifications). Thus similar techniques might apply. The problem is that modification is likely to be more dynamic, so the particularly static techniques we used might not work out as well.

More generally, the exact limits of applicability of pointer type annotations in optimization are not clear. Certainly source to source transformation is limited by the target language in its expressiveness. For example, address calculation such as that required for array subscripts cannot be expressed directly in Fortran, though it can be in C, etc. In any case, perhaps the point is that the technique works for us in this application.

References

[ACC82] Malcolm Atkinson, Ken Chisolm, and Paul Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Not.*, 17(7):24–31, July 1982.

[BBB+88] Francois Bancilhon, Gilles Barbedette, Véronique Benzaken, Claude Delobel, Sophie Gamerman, Cristophe 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 Persistent Programming Research Project 13, University of St. Andrews, Scotland, 1986.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[CDG+88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical Report ORC-1, DEC Systems Research Center/Olivetti Research Center, Palo Alto/Menlo Park, CA, 1988.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report DEC SRC 52, DEC Systems Research Center/Olivetti Research Center, Palo Alto/Menlo Park, CA, November 1989.

[CDG+91] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. In Nelson [Nel91], chapter 2, pages 11–66.

[CM84] George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 316–325, Boston, Massachusetts, June 1984. *ACM SIGMOD Rec. 14*, 2 (1984).

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

[Diw91] Amer Diwan. Stack tracing in a statically typed language, October 1991. Position paper for OOPSLA '91 Workshop on Garbage Collection.

[DMH92] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. SIGPLAN, ACM Press.

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

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

[Har92] S. P. Harbison. *Modula-3*. Prentice Hall, New Jersey, 1992.

[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 Nineteenth 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. COINS Technical Report 91-25, University of Massachusetts, Amherst, MA 01003, 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 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 Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119, Asheville, NC, December 1993.

[HMB90] Antony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an object faulting persistent Smalltalk. COINS Technical Report 90-45, University of Massachusetts, Amherst, MA 01003, May 1990.

[HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *ACM SIGPLAN Not. 27*, 10 (October 1992).

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

[Kae86] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. In OOPSLA [OOP86], pages 87–106.

[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.

[Mos87] J. Eliot B. Moss. Implementing persistence for an object oriented language. COINS Technical Report 87-69, University of Massachusetts, Amherst, MA 01003, September 1987.

[Mos89] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 269–285, Gleneden Beach, Oregon, June 1989. Morgan Kaufmann. Also available as COINS Technical Report 89-68, University of Massachusetts.

[Mos90] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, April 1990.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[MS88] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In Dittrich [Dit88], pages 298–316.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, New Jersey, 1991.

[OOP86] *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, September 1986. *ACM SIGPLAN Not. 21*, 11 (November 1986).

[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, Computer Sciences Department, University of Wisconsin, Madison, WI, 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.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In OOPSLA [OOP86], pages 9–16.

[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 *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 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.

[SRA94] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[WD92] Seth J. White and David J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth 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 Press.