Towards Compile-Time Optimisations for Persistence[*]

Antony L. Hosking
hosking@cs.umass.edu

J. Eliot B. Moss
moss@cs.umass.edu

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

We consider how a persistent programming language might offer performance competitive with that of non-persistent languages, at least on memory resident data. We are concerned with object-oriented languages, and with implementing persistence via *object faulting*, where the system detects uses of non-resident objects and fetches them on demand. We present some background on object faulting and means for implementing it, and describe a specific language we are developing, namely Persistent Modula-3. Then we explore approaches to optimising persistence aspects of Persistent Modula-3, and outline techniques under consideration in our compiler development effort.

1 Introduction

The Object Oriented Systems Group at the University of Massachusetts is engaged in research exploring the integration of programming languages with database technology. As a part of this work, we are involved in the development of *persistent programming languages*, as popularised by the PS-Algol effort [Atkinson *et al.*, 1981; Atkinson and Morrison, 1985].

We approach the problem of language-database integration from the object-oriented standpoint. Our particular approach to persistence is to integrate existing programming languages with our own persistent object store, Mneme [Moss and Sinofsky, 1988; Moss, 1989]. While there is some difference in functionality between Mneme and other object storage systems such as the Exodus storage manager [Carey *et al.*, 1986; Carey *et al.*, 1989] or Observer [Skarra *et al.*, 1987; Hornick and Zdonik, 1987], our integration techniques would extend to them.

We have chosen two object-oriented languages for integration with Mneme: Smalltalk [Goldberg and Robson, 1983] and Modula-3 [Cardelli *et al.*, 1989]. Our reasons for choosing these are as follows. First, they are relatively well-known. Smalltalk was the first object-oriented programming language to gain widespread recognition. Modula-3 is less widely known, but its ancestry is quite familiar, as it derives from the class of languages including Pascal [Jensen and Wirth, 1974], Modula-2 [Wirth, 1983], and Oberon [Wirth, 1988a; Wirth, 1988b]. Second, they represent quite different philosophies. In Smalltalk there is no static type checking. In fact, the Smalltalk class hierarchy provides only a very weak notion of type. Calls are bound at run time using a hierarchical method lookup mechanism, based on the class of the object on which the method is being invoked. If the lookup fails then a run-time error is signalled, indicating that the method is undefined for that class. On the other hand, while retaining dynamic binding of methods, Modula-3 does guarantee type-correctness of programs at compile time—the compiler will detect the invocation of methods that are undefined, or whose arguments are of the wrong type. By choosing two such different languages we hope to explore the generality of our techniques, and identify alternatives arising out of their differences.

When a program manipulates persistent data a decision must be made as to when that data should be fetched from stable storage into memory. One extreme is to require all persistent data to be made memory resident before the program begins manipulating any of it. In the case that a program accesses only a small fraction of the persistent data, such indiscriminate preloading is clearly undesirable.

[*]To appear in the Proceedings of the Fourth International Workshop on Persistent Object Systems: Design, Implementation, and Use (Martha's Vineyard, MA, Sept. 1990). This project is supported by National Science Foundation Grants CCR-8658074 and DCR-8500332, and by Digital Equipment Corporation, GTE Laboratories, and the Eastman Kodak Company.

Rather than preloading, we can dynamically check the residency of a persistent data item, fetching it if necessary. In this way the subset of all persistent data that is resident grows dynamically, as more and more data items are accessed by the program. This approach incurs a run-time overhead in that each access to a persistent data item requires an explicit residency check, resulting in its retrieval if it is non-resident. In the case where a persistent data item is always accessed by dereferencing some kind of *pointer* to it, this approach reduces to a technique that we call *object faulting*. A residency check is performed every time a persistent pointer is dereferenced, resulting in an *object fault* to retrieve the data item if it is non-resident. We can consider all such referenced data items (both persistent and volatile) as a kind of virtual heap. Object faulting does not preload the entire heap; rather, objects are faulted on demand.

Our design of Persistent Smalltalk makes object faulting the responsibility of the run-time system [Hosking *et al.*, 1990]. All references to non-resident objects are trapped by the Smalltalk virtual machine. Certain heuristics are used to restrict residency checks to just the send bytecodes and some primitives. This approach is purely dynamic in that no changes are made to the Smalltalk compiler or image. Our approach with Modula-3 is to have the *compiler* generate in-line code to perform the residency check, along with a call to the object fault handler if the check fails, wherever a persistent pointer is dereferenced. Note that in contrast with Smalltalk, this does not impose the burden of residency checks on the run-time system. Rather, the compiler statically determines where the checks will occur.

Of particular interest to us is that our persistent programming languages exhibit good *performance*. There are two aspects to the performance of a persistent program. We have already mentioned that persistence usually implies some sort of residency check to ensure data items are memory resident before their contents are accessed. We would like to minimise this overhead, so that in the case that all of its data is resident, a persistent program can approach the performance of its non-persistent counterpart. The second aspect of performance pertains to the *fetching* of non-resident data from stable storage. In this case, good performance is more difficult to define, but can be characterised as that which makes the best use of system resources. To sum up, there are two costs to persistence:

- the cost of residency checks, and
- the cost of fetching, storing, and managing non-resident data.

In this paper we focus on mitigating the cost of residency checks, through the application of what are typically considered to be compiler techniques. Addressing the second aspect of performance would seem to require techniques most often used in databases to cluster data for retrieval.

Our approach to improving performance is to devise and exploit compile-time optimisations that will eliminate or circumvent residency checks. With this emphasis on static techniques we concentrate on their realisation for Modula-3, which submits to stronger compile-time analysis than Smalltalk, and for which we must already modify a compiler to support persistence. This approach is complicated by the dynamic call binding that occurs with object methods in Modula-3—methods are bound to an object when it is created. Such late binding means that at any particular call site the actual code to execute cannot be statically determined. This prohibits such standard optimisations as inlining of method code.

The contributions of this paper are as follows. First, we have begun to frame the issues regarding the *performance* of persistent programming languages. Second, we offer several approaches to implementing persistence, and identify some potential performance problems with those approaches. Finally, we have begun to identify compile-time optimisations for bringing the performance of some aspects of persistent programming languages very close to the performance of their non-persistent counterparts.

The remainder of the paper is organised in the following way. We first review object faulting and some of the requirements it imposes, and then consider techniques for implementing object faulting. Following this background, we describe the Modula-3 programming language, the changes we are making to move from Modula-3 to Persistent Modula-3, and sketch some straightforward techniques for implementing Persistent Modula-3. Finally, we consider compile-time optimisations for improving the performance of persistence, using Persistent Modula-3 as a specific language for concreteness.

2 Requirements for Object Faulting

We have indicated that object faulting requires that there be a mechanism for checking the residency of an object. While we would wish that this check be as cheap as possible we assume that it does incur some marginal cost. That is, we do not assume any special hardware support, for example from paging hardware. Furthermore, so that residency checks may be machine independent we do not assume any particular hardware architecture.

Given that there is *some* mechanism for detecting the need for an object fault, we must have some way of handling the fault. We assume the existence of an underlying *object manager* that, given some unique *object identifier*, will return a pointer to the object in its buffers, retrieving it from the persistent object store if necessary. There is an issue here of format mismatch between objects as they are represented in the store and objects as they are represented in memory. This is especially significant when an object contains references to other objects. These references will typically be represented as object identifiers in the store, but may need to be converted to memory pointers when the object is made resident. This conversion process, known as *swizzling*, can be performed in two ways: *in-place* or by *copying*. In-place swizzling simply overwrites the object in the object manager's buffers with the converted version of the object. This requires that all objects in a buffer be *un*swizzled before the buffer is written back to disk. Unswizzling is complicated by the fact that some of the objects in the buffer may now have pointer references to volatile objects in the heap. These objects and all objects accessible from them must be made persistent.

Copy swizzling maintains a separate converted version of the object in the volatile part of the heap, corresponding to the unconverted version in the object manager's buffers. There is some cost to maintaining this correspondence since every update to the in-heap version of the object eventually results in a corresponding update to the buffer version, and may involve making some volatile objects persistent, as for in-place unswizzling.

The relative advantages of each scheme are obvious: in-place swizzling demands less memory, but may involve more work when unswizzling. Copy swizzling does have the added benefit that buffers may be removed more easily, since only modified objects must be unswizzled. Of more importance is the fact that some persistent object stores will not allow in-place swizzling since applications are denied access to objects in the store's buffers.

One aspect of buffer removal that we have not yet addressed is that any direct pointers to buffer objects from other parts of the heap must be located and updated to reflect the fact that the objects they refer to are no longer resident. Locating the pointers requires that a *remembered set* be maintained for each buffer, indicating all memory locations that contain pointers into the buffer. Whenever a pointer to an object in the buffer is stored in some memory location, we must check to see if the buffer's remembered set should be updated to reflect the store. Techniques such as this are employed by *generation scavenging* garbage collectors [Ungar, 1984]. Such garbage collectors have been shown to have superior performance for interactive systems such as Smalltalk [Ungar, 1987]. Given this performance reputation we assume that generation scavenging will be the garbage collector of choice, so that maintaining remembered sets for the buffers of the object manager will require little additional mechanism. One unfortunate drawback of this is that there will always be the overhead of performing store checks; they cannot be eliminated. However, the overall performance of generation scavenging for persistence is not known at this point in time. In systems such as Smalltalk remembered sets do not seem to grow very large. For persistence, remembered sets will probably be much larger (since it is highly likely that more data will be persistent than volatile). How larger remembered sets will affect the performance of generation scavenging remains to be seen.

3 Techniques for Fault Detection

In the previous sections we have identified the need for some mechanism to check the residency of an object, to detect when an object fault should occur. The object manager handles a fault by returning a direct memory pointer to the desired object, first retrieving it from the store if it is non-resident. Here we elaborate on how residency checks may be performed. Let us consider the virtual heap to be a *directed graph*. The *nodes* of the graph are the objects, and the *edges* of the graph are the references from one object to another. A computation traverses the object graph, which is only partially resident in memory. Traversing an edge from a resident object to a non-resident object causes an object fault, and the link is *snapped* to point to the resident object. It is important to note here that merely naming an object does not cause an object fault. Only when the *contents* of the object need to be accessed, and so the link to the object must be traversed, is it required that the object be made resident.

We need to be able to detect the traversal of a link from a resident object to a non-resident object. There are effectively just two ways of achieving this:

- *Mark the edges* of the graph that are links to non-resident objects, distinguishing them from links to resident objects (see Figure 1(a)).
- *Mark the nodes* of the graph to distinguish resident objects from non-resident objects (see Figure 2(a)).

Edge marking is relatively easy to implement by tagging pointers. Checking whether a pointer refers to a resident object or not is simply a matter of checking the tag. When a marked link is traversed, an object fault occurs and is

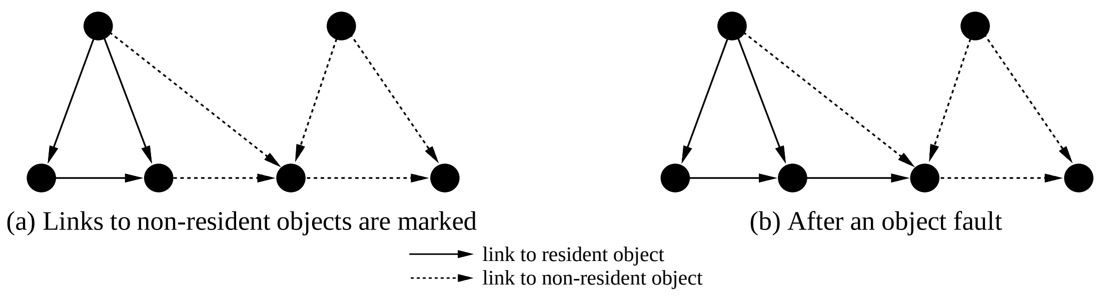
(a) Links to non-resident objects are marked
(b) After an object fault
link to resident object
link to non-resident object

Figure 1: Edge Marking

handled by the object manager, which returns a pointer to the resident object. The marked link is then snapped to point to the resident object (see Figure 1(b)). Note that it is legal (though suboptimal) for a marked edge to refer to a resident object, but an unmarked edge may never refer to a non-resident object.

Node marking is complicated by the fact that the non-resident objects are just that, non-resident, and must (paradoxically) be in memory for them to be checked. Solving this problem is simple. Specially marked resident pseudo-objects called *fault blocks* stand in for non-resident objects. Every reference from a resident object to a non-resident object is actually a pointer to a fault block (see Figure 2(b)). When a link is traversed to a fault block a fault occurs and is handled by the object manager. "Snapping the link" in this case involves setting the fault block to point to the object in memory (see Figure 2(c)). Note that there is now a level of indirection via the fault block; this may be bypassed by also updating the traversed link to point to the object in memory (see Figure 2(d)).

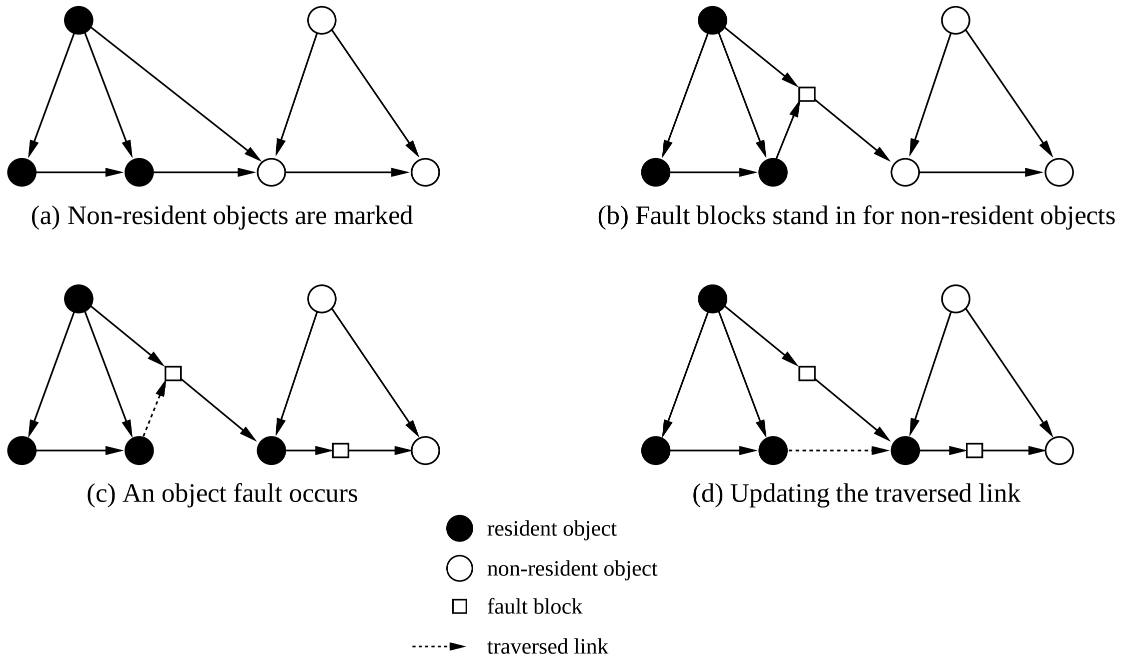
(a) Non-resident objects are marked
(b) Fault blocks stand in for non-resident objects
(c) An object fault occurs
(d) Updating the traversed link
resident object
non-resident object
fault block
traversed link

Figure 2: Node Marking

The preceding discussion assumes that we have an object identifier available to present to the object manager to handle a fault. In the edge marking scheme we store this identifier in the bits of the pointer that are not reserved for the tag. In the node marking scheme we store it in the fault block.

Each of these schemes has its particular advantages and disadvantages. In node marking, a particular fault block may be referenced by many resident objects. This means that the object manager need only be called once per fault block, to obtain a memory pointer to the corresponding object, when the first link to the fault block is traversed. The memory address is then cached in the fault block so that subsequent traversals of links to that fault block can pick it up from there, without additional calls to the object manager. However, there are overheads associated with fault blocks: storage and management for fault blocks; creation of fault blocks for all the objects referred to by a non-resident object when it is swizzled; and the extra level of indirection that fault blocks imply.

Edge marking has the advantage of eliminating the space consumed by fault blocks, and the level of indirection associated with them. Its disadvantage is that the only link that is "snapped" when an object fault occurs is the link that is traversed. All other links to the object are still marked as pointing to a non-resident object. This means that every traversal of a marked link will result in a call to the object manager to determine the object's address, regardless of whether the object is already resident or not.

Snapping the link has been mentioned as one way of reducing the overhead of object faulting. In the edge marking scheme this will usually be of little benefit, since the marked link that ends up getting snapped will probably be in a register or perhaps some other temporary location. The compiler may be of some help here if it can statically determine the source of the marked reference, making sure that it also gets updated when the link is snapped.[1]

4 Modula-3

Modula-3 consists primarily of Modula-2 with extensions for threads (lightweight processes in a single address space), exception handling, objects and methods, and garbage collection, while dispensing with variant records, and the ability to nest modules. Exception handling and threads do not raise any novel issues, so we will not discuss them further. Understanding the remaining extensions requires some understanding of the type system of Modula-3.

Modula-3 is *strongly-typed*: every expression has a unique type, and assignability and type compatibility are defined in terms of a single syntactically specified subtype relation, written `<:`. If `T` is a subtype of `U`, then every instance of type `T` is also an instance of type `U`. Any assignment satisfying the following rule is allowed: a `T` is assignable to a `U` if and only if `T` is a subtype of `U`. In addition there are specific assignment rules for ordinal types (integers, enumerations, and subranges), references (pointers), and arrays. We discuss only the specifics of reference types here.

A reference type may be *traced* or *untraced*. A traced reference (of type `REF T`) refers to storage (of type `T`) that is automatically reclaimed by the garbage collector whenever there are no longer any (traced) references to it. Untraced references (of type `UNTRACED REF T`) are just like Pascal pointers—the storage they refer to must be explicitly allocated and deallocated. The type `REFANY` contains all traced references, while `ADDRESS` contains all untraced references. The type `NULL` contains only the reference value `NIL`.

Object types are also reference types. An *object* is either `NIL` or a reference to a data record paired with a method suite, which is a record of procedures that will each accept the object as a first argument. Since they are references, objects may also be either traced or untraced. Every object type has a supertype, *inherits* the supertype's representation and implementation, and optionally may extend them by providing additional fields and methods, or overriding the methods it inherits with different (but type correct) implementations. When an object is created, one may supply specific methods for that individual object (again, they must be type correct), overriding the default implementations supplied by the object's type.

This scheme is designed so that it is (physically) reasonable to interpret an object as an instance of one of its supertypes. That is, a subtype is guaranteed to have all the fields and methods defined by its supertype, but possibly more, and it may override its supertype's method implementations with its own. In addition, an object's method values are not determined until the object is allocated, although the values cannot be changed after that.

An object type is specified by the following syntax:

```
T OBJECT fields METHODS methods END
```

This specifies an object subtype of `T`, with additional fields `fields` and additional or overriding methods `methods`. An object inherits its traced-ness from its supertype. There are two built-in object types, one traced and the other untraced, having no fields or methods, from which all object types are descended: `ROOT` and `UNTRACED ROOT`.[2]

We can summarise the subtype rules for references as follows:

[1] We may also apply certain heuristics to this problem. For details see [Hosking *et al.*, 1990].

[2] Shorthands `OBJECT...END` and `UNTRACED OBJECT...END` may be used for the forms `ROOT OBJECT...END` and `UNTRACED ROOT OBJECT...END`, object types inheriting from `ROOT` and `UNTRACED ROOT`, respectively.

```
NULL <: REF T <: REFANY
NULL <: UNTRACED REF T <: ADDRESS
NULL <: T OBJECT...END <: T, for some object type T
ROOT <: REFANY
UNTRACED ROOT <: ADDRESS
```

Finally, for garbage collection we must be able to find all references to traced data. This means that an *untraced* data item cannot contain any *traced* references. For this reason, records and arrays containing traced references are implicitly traced. These restrictions are summarised in the following table:

$\nearrow$	untraced	traced
untraced	$\surd$	$\times$
traced	$\surd$	$\surd$

That is, a traced data item may contain both traced and untraced references, but an untraced data item may contain only untraced references.

Let us briefly consider an implementation for objects. Because an object can be interpreted according to many different types, it must somehow carry a type code with it so that we can tell what its actual type is. Further, since the methods vary by object type, and possibly even by object, we need some way to find the methods when they are invoked. The expected implementation is for the object fields to be preceded by a pointer to the method suite. The method suite is simply a vector of addresses of procedures, preceded by the type code. Since the offset of a given method within the method suite is static, no run-time search is required to find the code to run on method invocation. Similarly, field offsets are statically known. This implementation approach is illustrated in Figure 3.

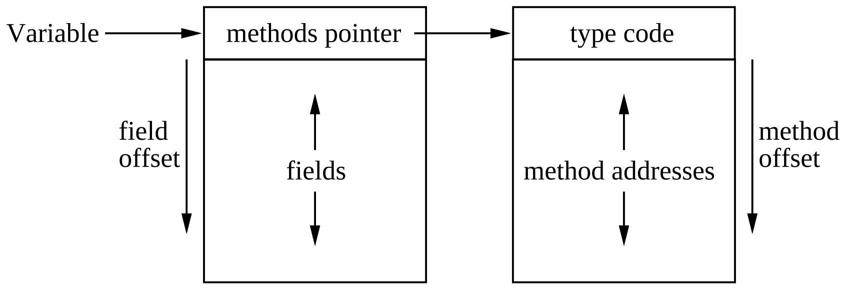

Figure 3: An implementation of Modula-3 objects

5 Persistent Modula-3

The previous section gave an introduction to the Modula-3 type system. In this section we extend that type system to incorporate persistence by adding a third class of reference types: *persistent* references, similar to the **db** types of the E database programming language [Richardson and Carey, 1987]. A persistent reference type is indicated by the keyword `PERSISTENT`, analogous to `UNTRACED`. A persistent reference indicates a specific object, but that object may or may not be resident in memory. In addition, we permit any top-level variable[3] to be qualified by the `PERSISTENT` keyword, since a program must have at least one persistent data item to start with, from which other persistent data can be reached. These may be thought of as implicit persistent references to known root objects in the persistent store.

Our new reference types have subtype rules analogous to traced reference types as follows:

```
NULL <: PERSISTENT REF T <: PERSISTENT REFANY
PERSISTENT ROOT <: PERSISTENT REFANY
```

Once again, an object inherits persistence from its supertype. A variable at top-level may be declared persistent using the `PERSISTENT VAR` construct just as `VAR` is used for non-persistent variables.

[3] A top-level variable is a variable declared in the outermost scope of a module.

Finally, analogous to traced data, anything reachable from a top-level persistent variable via persistent references will itself persist, so that we must be able to find all references to persistent data. Thus, an untraced data item may not contain persistent references. Whether a persistent data item may refer to non-persistent data is another question. At first glance this may seem to be a little strange, since the data referred to by the persistent data item will disappear when the program ceases execution, leaving dangling references. However, we can give such references special semantics, allowing them to be used to refer to volatile data while the program is running, but setting them to `NIL` when the persistent data item is written back to stable storage (or when it is loaded from stable storage). We augment the table from the previous section to summarise these rules:

$\nearrow$	untraced	traced	persistent
untraced	$\surd$	$\times$	$\times$
traced	$\surd$	$\surd$	$\surd$
persistent	†	†	$\surd$

† $\equiv$ special semantics

This design is admittedly non-orthogonal: orthogonality would require that any data item be able to contain persistent references and, vice versa, every persistent data item be able to contain references to volatile data, making them persist by transitivity. However, non-orthogonality *is* consistent with the traced/untraced distinction, and as with the overhead of garbage collection, allows programmers to indicate explicitly whether or not they accept the overhead of persistence. Furthermore, it makes it easier for us to perform experiments to evaluate the relative performance of using persistent references over ordinary references. Later on we could collapse the persistent and traced distinction, as is done in Smalltalk where all references are potential references to persistent data. Whether or not we do this will depend on performance—if persistence imposes little performance degradation then there is no need for the distinction.

6 Implementation

We have indicated the syntax and semantics of persistent types and variables for Modula-3. Now we turn to the straightforward implementation of these extensions, using the techniques of object faulting. For `PERSISTENT REF` types the simplest implementation is to perform a residency check every time a given reference is used. We may use either of the node or edge marking schemes. Node marking simply implies the use of fault blocks, marked with some tag bit to enable the check. The overhead of the extra indirection can be removed later by the scavenger: if it detects a reference to a fake object that has a real object attached to it, the reference is updated to point to the real object. The fault block's storage may eventually be reclaimed.

Edge marking is more of a problem, since we need to be able to tag persistent references, to enable the check. On byte-addressed architectures this can be achieved by ensuring that all persistent data is word-aligned, leaving a low-order bit free to be used as the tag. Alternatively, we might use the sign bit to distinguish memory pointers from persistent object identifiers. Both of these techniques make certain assumptions about machine architecture. In some cases tagging may simply be impossible.

We can implement field access for persistent `OBJECT` types similarly to `PERSISTENT REF` types. For method invocation, however, we can use the following technique to eliminate conditional code in the residency check. Given that we have a fault block standing in for the resident object, we can supply a fake method suite for the fault block. The fake method suite would contain only procedures that will fault in the real object. At fault time, when the fault block is updated to point to the real object, we would also update the fault block's fake method suite to forward calls to the real object. This technique could also be used for field access if we are prepared to turn field access into method invocation.

As for `PERSISTENT VAR`, we can treat every persistent variable as if it is an implicit `PERSISTENT REF`, and use the same implementation techniques.

We have not yet mentioned that the methods (code) of persistent objects must also persist. Clearly ordinary code will persist by virtue of the fact that it is in some program on disk. The question here is whether the code implementing the methods should reside in the persistent object store instead of in the program. Ideally, objects should carry their methods with them wherever they go, even if they are used in a different program from the one that created them. For this, method code must reside in the store. We then face the problem of *dynamically* linking method code with a running program. This does not appear to present any fundamental difficulties, but we have no specific design at this time.

7 Optimisations

The previous section indicated a straightforward implementation of persistence for Modula-3. In this section we look at improving the performance of persistent programs by using compile-time optimisation techniques to eliminate residency checks. Assuming that a resident object is never made non-resident (during a program's execution), then performing a residency check once for a given reference is as good as performing it many times. Further occurrences of the check are thus superfluous and may be eliminated. To eliminate checks we can apply the traditional static analysis techniques used by compilers. We first discuss optimisations that may be enabled by both local (basic block) and global (intra-procedural) analysis, to eliminate redundant residency checks within a procedure. Then we consider less traditional optimisations, requiring inter-procedural analysis, that make use of co-residency properties of persistent data items to amalgamate their residency checks into just one check. Two data items are said to be *co-resident* if, whenever one of them is resident, then so is the other.

Local optimisations are those that may be performed by examining only the statements within a basic block (a section of straight-line code having just one entry point and one exit point). A typical local optimisation is common subexpression elimination. We can use a similar approach to eliminate redundant residency checks within a basic block. If we consider a residency check as evaluating a boolean expression, then the first residency check performed for a reference makes the expression true. All later checks can be replaced by the value TRUE.

This approach will extend to global (intra-procedural) analysis. Furthermore, global analysis enables other optimisations such as dead code elimination and code motion. Given that we can reduce some residency checks to TRUE, then the fault handling code (which is executed when the check evaluates to FALSE), becomes dead and may be eliminated. Also, code motion can be used to move a loop invariant residency check out of a loop. Code hoisting can be used to replace two residency checks that occur in different paths of the program by just one. Techniques similar to these were shown to be quite effective in E [Richardson, 1989].

Procedure inlining replaces a full call to a procedure with the code that implements that procedure. This allows the called code to be optimised in the context of the call site, integrating its analysis with that of the calling procedure. However, inlining is more difficult for methods, because dynamic binding implies that in general we cannot know which specific method will be invoked. It is clear that allowing objects to override their type's default methods at creation time poses some difficulty. Rather than eliminating the feature from the language, we could make use of *pragmas* inserted by the programmer in an object type declaration indicating that a particular method will never be overridden by an individual object instance of that type. Let us consider inlining of a method call in three particular cases. In the first, suppose that data-flow analysis reveals the exact type of the object on which the method is being invoked, and that we can determine (either by pragma or analysis) that the object does not override the type's default implementation for that method. Then we can inline the default code. In the second, we assume that the object's exact type is known, but that it may override the method. We can still inline the default code, preceding it with a check of the object's method suite to make sure that the method's slot contains the default method), and generate code to do a full call in the failure case. In the third case we make no assumptions, but replace the call with conditional code, branching on the type of the object, and inline the default methods as for the second case. This technique is known as *message splitting* and has been applied in other object-oriented languages such as SELF [Chambers and Ungar, 1989; Chambers and Ungar, 1990].

If we are prepared to do some inter-procedural analysis, we can use *customised compilation* to tailor the compilation of a procedure to the characteristics of a particular call site. Just as inlining allows the compiler to optimise a procedure in the context of a particular call site, so does customised compilation, but without the additional space cost that inlining imposes, since the customised version is out of line. The customised compiled procedure can only be used at call sites having appropriate characteristics. Again, this has been used previously in SELF and also in Trellis[4]/Owl [Schaffert *et al.*, 1986]. Applying customised compilation to persistence, we can compile customised versions of procedures based on assumptions about the residency of their arguments. A particular customised version can then be used at any call site where its residency requirements are satisfied.

All of these techniques make use of information gleaned from the *code* at compile-time. Going back to our analogy of the virtual heap as a directed graph, the code establishes possible *traversals* of the graph. We can also derive

[4] Trellis is a trademark of Digital Equipment Corporation.

information from the *types* as to the potential *structure* of the graph. If we have co-residency information available attached to the types, then we can *amalgamate* the retrieval of data items that are indicated as being co-resident, reducing the number of residency checks needed. We can best explain this with an example. Consider the following persistent reference type declaration:

```
TYPE
  PRecord = PERSISTENT REF RECORD
    field1 : PERSISTENT REF foo;
    field2 : PERSISTENT REF bar;
  END;
```

Suppose that we know that a `PRecord` should be co-resident with the `foo` object named by `field1`. Then, whenever we fault a `PRecord` data item we simultaneously fault the `foo` item, and ensure that the `PRecord` points directly to the `foo` item. Therefore, the compiler does not need to generate a residency check for uses of `field1` components of a `PRecord`.

One way to represent co-residency information is via annotations to the *type graph*. The nodes of this graph are the types of the program, and the edges indicate uses of one type in defining another. In the above example, there would be nodes for the types `PRecord`, `foo`, and `bar`, and edges from `PRecord` to `foo` and `PRecord` to `bar`. To indicate co-residency we mark edges of the type graph. For example, we could mark the edge from `PRecord` to `foo`, to indicate the co-residency assumption discussed above. In order to provide more precise information we can distinguish edges based on the name of the record component, etc., to which they correspond.

Of course, there is the question of where the co-residency information comes from. One option is to allow programmers to annotate their type declarations with pragmas indicating desired co-residency properties. More preferable would be the automatic derivation of these properties. Static analysis of a procedure can determine some information about desirable co-residency properties for types used in the procedure. For example, intra-procedural analysis can say whether a path through the type graph will *definitely* be traversed, *may* be traversed, or is *never* traversed, when executing the procedure. It is not obvious how to combine analyses of individual procedures to annotate a global type graph with co-residency assumptions. Even so, a global type graph might be desirable, since it would allow the same assumptions to be made throughout the program. This would ensure that optimisations are applied consistently. Further, the global type graph is a relatively simple data structure for the run-time system to use in enforcing co-residency assumptions when handling object faults.

The global type graph is not without problems, though, since each assumption represents a compromise between procedures that follow long paths through the type graph and procedures that follow short ones. If a procedure follows shorter paths, then the global type graph assumptions may cause unnecessary fetches. If a procedure follows longer paths, then it will need to perform more residency checks.

Another problem with the global type graph approach is that a single object may have an unbounded number of objects required to be co-resident with it. For example, large, homogeneous data items such as arrays have many references to the same type, causing a combinatorial explosion because of the potentially high branching factor. Recursively defined types also pose difficulties since they create cycles in the type graph, and thus introduce co-residency paths of unbounded length. For example, consider the singly-linked list type:

```
TYPE
  SLList = PERSISTENT REF RECORD
    next : PERSISTENT REF SLList;
    ...
  END;
```

If the `next` field is marked for co-residency, then fetching any item in the list will fetch all subsequent items.

The shortcomings of the single global type graph may be overcome by putting each use of a type in context. For example, we might specialise the annotation on a type for each variable of the type. We can view this as annotating the variables instead of the types. Each variable has attached to it a subgraph of the full type graph; the root of the subgraph is the variable's type. This subgraph indicates what co-residency assumptions can be made about data reachable from the variable.

So far we have only considered the use of statically obtained co-residency information. There is also the possibility of including statistics obtained by dynamic profiling. If the types define the possible structure of the virtual heap, and the programs define the potential traversals of that structure, then profiling can approximate the probability that a particular traversal will occur. This information can be fed back into the compiler so that more intelligent co-residency decisions can be made based on actual usage patterns. This style of optimisation is similar to that used in some database systems for tuning indexing, clustering, etc.

We have briefly sketched techniques to determine when a number of potential faults can be merged together, so that they each share just one residency check. An interesting further use of co-residency characteristics would be to communicate them to the object manager for *clustering* purposes. Clustering places objects physically close together on disk, so that they may be retrieved with just one disk access. If data items that are co-resident can be clustered, then retrieving those items will be performed with fewer disk accesses. This seems to indicate the potential for even further gains, since it allows us to address the other aspect of performance with persistence: fetching, storing, and managing non-resident data. Even more interesting is the potential to turn things around. Whereas we have indicated that co-residency information may be used for clustering, we could have clustering information drive the co-residency analysis phase of the compiler. Co-residency analysis would still serve as input for *initial* data clustering, but thereafter decisions could be made using profiles of *entire suites* of programs. These clustering decisions could then be used in the compiler's co-residency analysis.

8 Conclusions

We have identified some of the issues regarding the performance of persistent programming languages, and introduced several approaches to implementing persistence. We have discussed how these approaches may be used in implementing Persistent Modula-3, through modification of the compiler. Finally, we have indicated possibilities for improvement of the performance of persistent programs through compile-time optimisations. Of particular interest is the role that co-residency analysis might play in eliminating residency checks and in obtaining clustering criteria for the underlying object store.

Acknowledgements

The Modula-3 implementation group at the University of Massachusetts consists of Rick Hudson and Amer Diwan, in addition to the authors. This group's regular design meetings generated many of the ideas for incorporating persistence in Modula-3.

References

[Atkinson and Morrison, 1985] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 539–559.

[Atkinson *et al.*, 1981] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices 17*, 7 (July 1981).

[Cardelli *et al.*, 1989] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Tech. Rep. DEC SRC 52, DEC Systems Research Center/Olivetti Research Center, Palo Alto/Menlo Park, CA, Nov. 1989.

[Carey *et al.*, 1986] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Databases* (Kyoto, Japan, Sept. 1986), ACM, pp. 91–100.

[Carey *et al.*, 1989] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage management for objects in EXODUS. In *Object-Oriented Concepts, Databases, and Applications*, Won Kim and Lochovsky Frederick H, Eds., Frontier Series. Addison-Wesley, ACM Press, New York, NY, 1989, ch. 14, pp. 341–369.

[Chambers and Ungar, 1989] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, OR, June 1989), vol. 24, no. 7 of *ACM SIGPLAN Notices*, ACM, pp. 146–160.

[Chambers and Ungar, 1990] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (White Plains, NY, June 1990), vol. 25, no. 6 of *ACM SIGPLAN Notices*, ACM, pp. 150–164.

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

[Hornick and Zdonik, 1987] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 70–95.

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

[Jensen and Wirth, 1974] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*, second ed. Springer-Verlag, 1974.

[Moss and Sinofsky, 1988] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In *Advances in Object-Oriented Database Systems* (Sept. 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–316.

[Moss, 1989] J. Eliot B. Moss. The Mneme persistent object store. COINS Technical Report 89-107, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Oct. 1989. Submitted for publication as "Design of the Mneme Persistent Object Store".

[Richardson and Carey, 1987] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementations in EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, CA, May 1987), vol. 16, no. 3 of *ACM SIGMOD Record*, ACM, pp. 208–219.

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

[Schaffert *et al.*, 1986] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), vol. 21, no. 11 of *ACM SIGPLAN Notices*, ACM, pp. 9–16.

[Skarra *et al.*, 1987] Andrea Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An object server for an object oriented database system. In *Proceedings of International Workshop on Object-Oriented Database Systems* (Pacific Grove, CA, Sept. 1987), ACM, pp. 196–204.

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, PA, Apr. 1984), ACM SIGPLAN Notices, ACM, pp. 157–167.

[Ungar, 1987] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

[Wirth, 1983] Niklaus Wirth. *Programming in Modula-2*, second, corrected ed. Springer-Verlag, 1983.

[Wirth, 1988a] Niklaus Wirth. From Modula to Oberon. *Software: Practice and Experience 18*, 7 (July 1988), 661–670.

[Wirth, 1988b] Niklaus Wirth. The programming language Oberon. *Software: Practice and Experience 18*, 7 (July 1988), 671–690.