Compiler Support for Persistent Programming*

Antony L. Hosking *J. Eliot B. Moss*

COINS Technical Report 91-25
March 1991

Object Oriented Systems Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

We present the design and implementation of Persistent Modula-3, a compiled, optimized, persistent programming language. The design allows the evaluation of performance aspects of persistent programming languages, while the implementation supports the development of compile-time mechanisms for *optimizing* persistent programs. We present several optimizations in detail, and outline further optimization opportunities.

*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.

1 Introduction

There is considerable current interest in the design of new programming languages that address the problems associated with the storage, retrieval, and manipulation of large amounts of highly structured, long-term data. The need for such support can be seen in the emergence of new classes of applications including computer-aided design (CAD), computer-aided software engineering (CASE), document preparation, and office automation. Traditional programming languages have failed to support these sorts of applications satisfactorily, since they provide only a very weak notion of long-term storage management in terms of the "file" data type.

Recently we have seen the emergence of *persistent* programming languages, that treat *persistence* as an important property of data. The designers of PS-Algol outlined a spectrum of persistence ranging up to and including data items that outlive the programs that create them [Atkinson *et al.*, 1983; Atkinson and Morrison, 1985]. They sought to make the longevity of a data item independent of the way it is manipulated, and conversely, the way programs are expressed independent of the longevity of the data they manipulate. In line with this principle, they also argued that persistence should be orthogonal to type, allowing all data items the full range of persistence.

Many programming languages incorporating some form of persistence have since been proposed and implemented. However, few have specifically addressed the issue of performance. Language designers have been interested in exploring the bounds of expressiveness and functionality in their languages, viewing implementation more as a means of validating the feasibility of the language design than showing that programs written in the language can be made to run efficiently. As the field has matured, the performance aspects of persistence have become more important. If persistent programming languages are to gain widespread acceptance then they must exhibit sufficiently good performance to attract programmers away from traditional programming languages.

Traditional languages have addressed the issue of performance in the compiler, performing substantial analysis at compile time to optimize programs. Persistence introduces a new set of optimization problems related to those found in database systems, resulting from the need to mitigate the high cost of manipulating a database that is only partially resident in memory.

Here we explore mechanisms for optimizing persistent programs in the context of a highly optimizing compiler for a persistent extension of Modula-3. The remainder of the paper is organized as follows. We first give an informal presentation of the syntax and semantics of Persistent Modula-3. Following this introduction to the language, we sketch its straightforward implementation using a technique we call *object faulting*. Finally, we present approaches to optimizing persistent programs so as to improve their performance.

2 Persistent Modula-3

Persistent Modula-3 is an extension of Modula-3 as defined by DEC/Olivetti [Cardelli *et al.*, 1989]. We have chosen Modula-3 as a vehicle for exploring optimization of persistent programs because it is strongly typed, compiled, and representative of a class of familiar languages, including Pascal [Jensen and Wirth, 1974] and Modula-2 [Wirth, 1983]. We are interested in exploring persistence in such a setting. Furthermore, strong typing permits optimizations that would be precluded in a more loosely typed language.

2.1 Modula-3

Modula-3 consists primarily of Modula-2 with extensions for threads (lightweight processes in a single address space), exception handling, generics, objects and methods, and garbage collection, while dispensing with variant records, and the ability to nest modules. Exception handling and generics do not raise any novel issues here, so we will not discuss them further. While threads do have implications for concurrency control and transactions in a persistent system, for the purposes of this paper we ignore these issues by treating an executing program as one transaction, whose changes to the persistent store are effectively committed when the program terminates. Understanding the remaining extensions requires some explanation 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. If T is a subtype of U (written $T <: U$), 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 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 some specifics of reference types here, omitting unnecessary details.

Reference types are *traced*. A 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 references to it. 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*: a record of procedures that will each accept the object as a first argument. Since they are references, objects are also traced. 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.

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

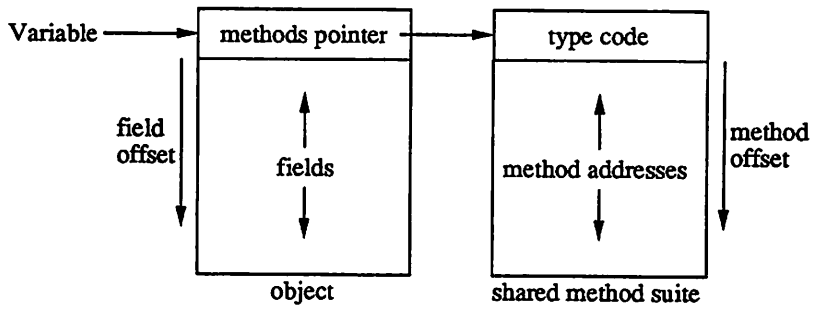

Figure 1: An implementation of Modula-3 objects

by its supertype, but possibly more, though it may override its supertype's method implementations with its own.

An object type is specified by the following syntax:

```
T object    fields
  methods methods
  override overrides
end
```

This specifies an object subtype of T, with additional fields fields, additional methods methods, and overriding methods overrides. There is a built-in object type, having no fields or methods, from which all object types are descended: **root**.

We can summarize the subtype rules for references as follows:

null <: **ref** T
null <: T **object**... **end** <: T, for some object type T

Let us briefly consider an implementation for objects. Because an object can be interpreted according to many different types, it must carry some sort of *type code* with it so that we can determine its actual type at run time (e.g., for garbage collection). Further, since the methods vary by object type, 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. This is simply a vector of addresses of procedures, preceded by the type code. Because Modula-3 offers single (not multiple) inheritance the offset of a given method within the method suite is static, so no run-time search is required to find the code to run on method invocation. Similarly, field offsets are statically known. The method suite for a particular object type is shared by all of its instances. This implementation approach is illustrated in Figure 1.

2.2 Extensions for persistence

To incorporate persistence in Modula-3 we have extended its type system with another 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**. A persistent reference indicates a specific data item, but that item may or may not be resident in memory. In addition, we permit any top-level variable (i.e., a variable declared in the outermost scope of a module) to be declared persistent using the **persistent var** construct, just as **var** is used for non-persistent variables. This indicates that the variable should be treated as a "handle" on a known data item in the persistent store. Our new reference types have subtype rules analogous to the ordinary reference types; an object type inherits persistence from its supertype.

Finally, analogous to garbage collecting traced data, anything reachable from a top-level **persistent var** via *persistent* references will itself persist, so that we must be able to find all persistent references. There is a question as to whether a persistent data item may contain non-persistent references. 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).

2.2.1 Binding Issues

There are a number of issues related to the binding of Modula-3 variables to their corresponding objects within the persistent store: top-level variables must be bound to their values; objects must be bound to their method code. We have indicated that top-level variables may be declared to be persistent, and that such variables act as "handles" on known objects within the store, from which other persistent data may be reached. The question remains as to when these known persistent objects are allocated and initialized. There is also the question of binding objects to their method code when they are made resident, raising the whole issue of code residing in the store.

2.2.2 Orthogonality Issues

The design we have just described yields non-orthogonal persistence: there is a separate hierarchy of persistent types, independent of the ordinary transient types. The type rules of Modula-3 prohibit the assignment of pointers from one hierarchy to pointer variables of the other hierarchy. In Modula-3 dynamic storage is allocated by the **new** statement:

ref := **new**(**ref** *T*);

This allocates storage for a traced data item of type *T*, and returns a transient reference to it. Since a **ref** *T* cannot be assigned to a variable of type **persistent ref** *T*, and persistence is based on reachability via persistent references, the potential longevity of a data item is determined when it is created—if a persistent reference is requested then the item may persist, otherwise it cannot.

A type orthogonal language design could be achieved by collapsing the distinction between ordinary and persistent references. However, we have reasons for maintaining the distinction. Rather than *designing* persistent programming languages, we are interested in exploring the performance ramifications of persistence. A non-orthogonal design serves this interest best since it permits easier performance evaluation. Clearly, dereferencing a persistent reference will in general be more expensive than dereferencing an ordinary reference, since it will involve a check to make sure that the data item referred to is resident. By maintaining the distinction, we are better able to compare the relative performance of persistent references, and get a better understanding of the overheads of persistence.

Another justification for maintaining the distinction may be found in Modula-3 itself. We have not yet mentioned that Modula-3 also provides another, separate, hierarchy of *untraced* references. These are just like Pascal pointers—the storage they refer to must be explicitly deallocated, rather than being reclaimed by the garbage collector. This allows programmers to indicate explicitly whether they accept the overhead of garbage collection. Similarly, a separate hierarchy of persistent references allows programmers to indicate explicitly whether they accept the overhead of persistence. If we can make this overhead small, then collapsing the distinction may be justifiable. This will likely depend on the quality of the optimizations we describe later in this paper.

3 Implementation

Our implementation of Persistent Modula-3 is inspired by the following requirements. We wish to support the language on a number of different hardware and operating system platforms. For this reason any implementation must be portable. Moreover, we are interested in exploring performance issues by evaluating different approaches to persistence, so the implementation should be flexible enough to allow variation.

There are two approaches we can take to detecting non-residency of persistent data:

- Use a hardware page protection scheme to trap references to non-resident data, so that residency checks are subsumed by ordinary memory access.
- Use a software scheme, thereby incurring some marginal cost in residency checks.

The chief advantage of the first approach is that there is no overhead for residency checks over and above the memory management overhead already in place. It also requires little additional mechanism. However, it does suffer from a number of disadvantages. First, there is sufficient variation between operating systems and hardware to pose a portability problem (e.g., some machines do not offer paging hardware). Second, many operating systems do not reflect page faults back to application programs particularly efficiently, so that performance may be compromised.

Third, the memory location of an object is fixed in advance, so that we cannot easily change the size of persistent objects, or move them. Finally, taking a hardware approach ties us to a particular strategy, preventing us from evaluating different approaches.

For these reasons we have devised a software scheme, called *object faulting*. By taking a software approach we assume some marginal cost associated with checking the residency of persistent data. Having accepted this cost, we would like to minimize its impact on the performance of our persistent programs. This will require reasonably sophisticated compile-time optimization of these programs.

We have chosen the GNU C compiler as a base on which to build, its portability meeting the first of our requirements, and its optimization phase allowing us to attack the problem of performance. Our implementation effort involves the construction of a Modula-3 front end to the compiler, along with the necessary modifications to the back end to support persistence and garbage collection.

3.1 Object Faulting

Object faulting is a mechanism that detects when a pointer to a non-resident object is dereferenced. An object fault causes the object to be retrieved from the persistent store, converted as needed, and made available to the running program in virtual memory.

Our approach to object faulting is similar to that used in LOOM [Kaehler and Krasner, 1983]. Specially marked resident pseudo-objects called *fault blocks* stand in for non-resident objects. Every memory reference to a non-resident object is actually a pointer to a fault block. In this way, pointers to non-resident objects may be distinguished from pointers to resident objects by checking whether they refer to a fault block.

The approach is illustrated in Figure 2. In Figure 2(a) we see that a non-resident object is referred to by pointers to its fault block, that contains sufficient information to locate the object on secondary storage. When a pointer to the fault block is dereferenced, a fault occurs, bringing the non-resident object into memory (Figure 2(b)). The fault block is overwritten with an *indirect block* that contains a pointer to the now resident object in memory. Note that the newly resident object may contain references to other persistent objects. These will typically be represented as object identifiers in the persistent store, but must be converted to in-memory pointers when the object is made resident. This process of converting objects from their persistent format to the in-memory format expected by the program is known as *swizzling*.[1] Note also that there is now a level of indirection via the indirect block. This overhead will eventually be eliminated by the garbage collector—any reference to an indirect block will be updated to point to the real object, and the space for the indirect block can then be reclaimed (Figure 2(c)).

[1] We are trying to determine the etymology of this term but have no definitive origin yet.

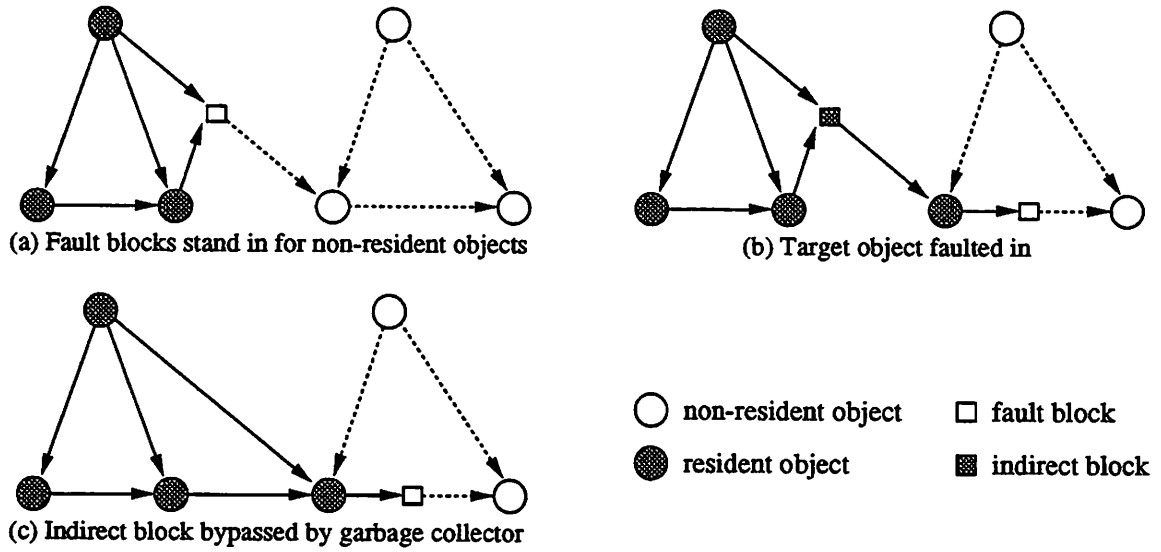

Figure 2: Object Faulting

3.2 Residency Checks and the Compiler

Implementing object faulting in Modula-3 is simply a matter of having the compiler generate code to perform a residency check wherever a **persistent ref** is dereferenced, and to handle the object fault if the residency check fails. The compiler can also ensure that the source of the reference that caused the fault is updated to point directly at the resident object, immediately bypassing the indirect block, rather than deferring to the garbage collector to bypass it later.

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 supply a fake method suite for the fault block. The fake method suite contains only procedures that fault in the real object before forwarding the call to the object's real methods. At fault time, when the fault block is overwritten with an indirect block that points to the real object, we set up another fake method suite for the indirect block to forward calls to the real object.[2]

3.3 Handling an Object Fault

Object faults are handled by calls to an underlying object storage subsystem. We use the Mneme persistent object store [Moss, 1990a], but we could just as easily use some other storage system such as the EXODUS storage manager [Carey *et al.*, 1986; Carey *et al.*, 1989]. Given a unique *persistent identifier* for an object, the persistent object store will return a pointer to that object in

[2]This technique can also be used for field access if we are willing to turn field access into method invocation; this is probably more expensive than a conditional residency check.

its buffers, retrieving it from secondary storage if necessary. These persistent identifiers are stored directly in the fault blocks, to be used at fault time.

Retrieving just one object at every object fault has been shown to be extremely inefficient. For this reason Mneme groups objects together into *segments* for retrieval. When one of the objects in the segment is to be made resident the entire segment is placed in a buffer in memory.

3.3.1 Swizzling

We have mentioned the need to swizzle objects from their persistent format to the in-memory format expected by the program. We perform *copy swizzling*[3] when an object is first faulted we make a swizzled copy of it in a specially managed persistent area of the heap. The persistent form of the object may contain references to other persistent objects. These references are typically stored as persistent identifiers, and must be converted to in-memory pointers. To avoid both conditional code and lookup cost in swizzling we convert all such references into newly allocated fault blocks. This means that there may be more than one fault block for any given object and implies that a fault block may refer to an object that is already resident. To avoid making more than one copy of a persistent object we must keep track of which objects have been swizzled by maintaining some sort of correspondence table mapping persistent identifiers to their swizzled copies; Mneme supports this mapping efficiently.

3.4 Writing Objects Back

When a program finishes execution, all of its persistent objects must be written back to disk. This means that the buffers of the storage manager must be updated to reflect any changes before the storage manager is asked to write those buffers out to secondary storage. Updating the buffers involves *un*swizzling all modified objects in the buffer: every in-memory pointer is replaced with the persistent identifier of the object it refers to. To enable this we store the persistent identifier with each swizzled copy. If a pointer refers to an object that has no persistent identifier stored with it, then the object is newly created and must be *promoted*: space must be allocated for it in the persistent store and it must be assigned a persistent identifier. In its turn it will also eventually need to be unswizzled, perhaps dragging further new objects with it into the persistent store.

If large amounts of persistent data are to be manipulated over the period of a program's execution it is likely that buffer management will need to be performed so that memory is not tied up with data that no longer needs to be resident. This means that we must be able to remove buffers from memory incrementally. In addition to unswizzling, we must decide what to do with the swizzled copies in the persistent area of the heap. The approach we take is illustrated in Figure 3. The

[3]For a performance evaluation of this and other swizzling techniques see [Moss, 1990b]

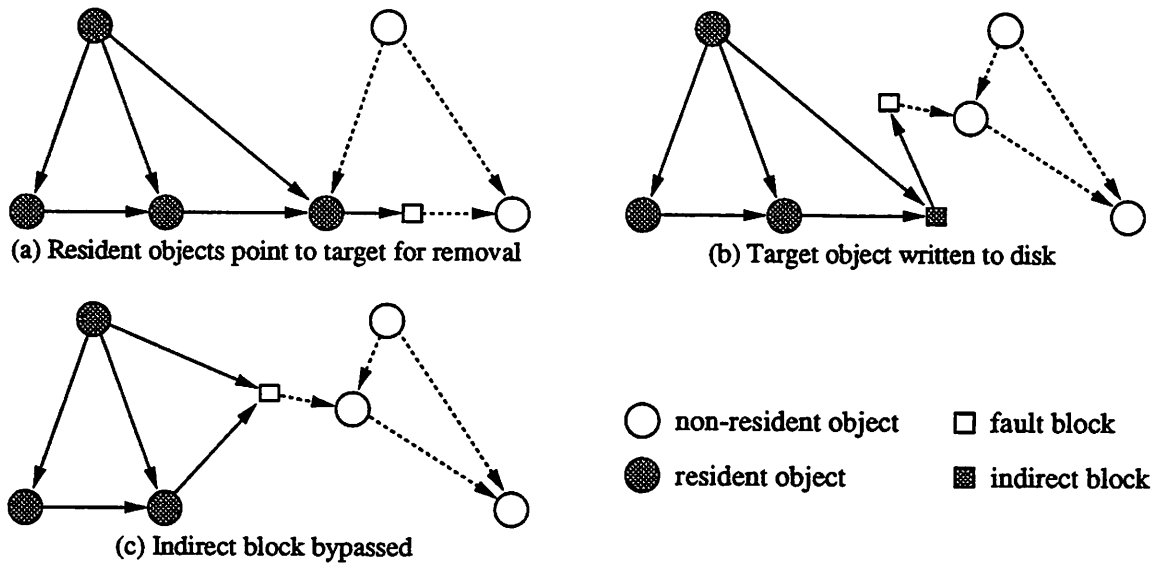

Figure 3: Removing Buffers

swizzled copy is overwritten with an indirect block that points to a newly allocated fault block for the object (Figure 3(b)). If the target object is accessed again in the future it will be faulted back into memory. As we shall describe, the extra indirection imposed by the indirect block allows us to reclaim the space that was used by the swizzled copies, after the indirect blocks have been bypassed (Figure 3(c)).

3.5 Managing the Persistent Area

We have mentioned that persistent objects are copy swizzled into a separately managed persistent area of the heap, separate from the volatile area in which all new objects, both transient and persistent, are allocated. The volatile area is garbage collected to reclaim free memory. In the presence of incremental buffer removal we would like to reuse any memory freed up in the persistent area of the heap. Here we describe our scheme for managing this area.

The persistent area is divided into chunks; these chunks are the unit of reclamation in the area. The life cycle of a chunk in the persistent area is illustrated in Figure 4. To reclaim a *live* chunk we process it as follows. First, we *close* the chunk so that no more swizzled persistent objects are allocated in it. Then we begin scanning the chunk, evacuating all objects in it, leaving indirect blocks in their place. To evacuate an object we can either unswizzle it into the storage manager's buffers—thence to be written back to disk as we have described—or we can copy it to some other live chunk. When this scan is finished we can mark the chunk as *evacuated*—there may still be pointers into the chunk from other parts of memory, but they will all point at indirect blocks. As we have mentioned earlier these indirect blocks will eventually be bypassed by the garbage collector.

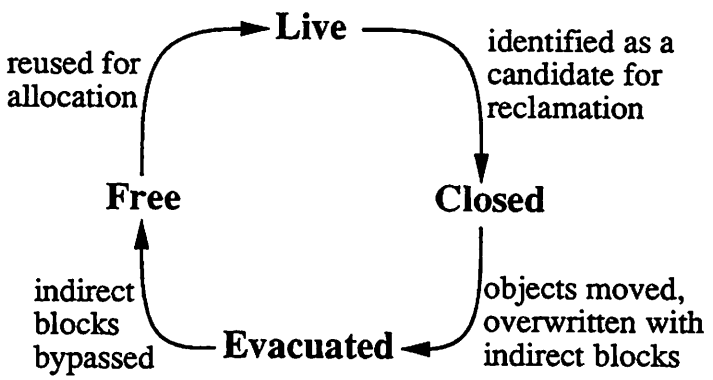

Figure 4: Managing the Persistent Area

In order to proclaim an evacuated chunk *free* we must ensure that there are no pointers into the chunk. We could scan the entire heap updating all pointers that refer to indirect blocks in evacuated chunks. To avoid scanning the entire heap we can keep track of all pointers into a chunk; for details of such an approach see [Ungar, 1984].

4 Optimizing Persistent Programs

The previous section indicated a straightforward implementation of persistence for Modula-3. In this section we look at improving the performance of persistent programs using compile-time optimization techniques. First we outline the costs incurred by persistence.

4.1 Costs of Persistence

We can decompose the cost of dealing with non-resident data into two components:

- *CPU costs*: the time spent detecting and managing non-residency. This includes the overheads of residency checks, swizzling, unswizzling, and promotion.
- *I/O costs*: the time spent accessing secondary storage (bringing data into memory and committing it to stable storage). This is influenced by the amount of data to be retrieved, the clustering of data into segments for retrieval, and the size of the available buffer space.

The first cost can be attacked using typical compile-time optimization techniques, such as global (intra-procedural) data flow analysis, to detect redundant residency checks. We can also make use of *co-residency* information to reduce residency checks. If a pair of data items can be guaranteed to be resident in memory at the same time then they are said to be co-resident. Reducing I/O costs implies the use of techniques typically found in database systems to cluster data for retrieval. There is some interplay between the two costs, since data flow analysis may reveal co-residency information about data, that may influence clustering decisions. Similarly, clustering information may affect what residency checks need to be performed.

4.2 Eliminating Residency Checks

Our implementation inserts a residency check (along with the fault handling call to the storage manager) wherever a persistent pointer is dereferenced. Let us assume that residency checks are idempotent: once a data item is faulted in then it remains resident. If this is the case then many residency checks will be redundant. Consider the following Modula-3 code:

```
type
 rec = record
        field: integer; ...
       end;
var
 ptr: persistent ref rec:= ...;
 x, y: integer;
begin
 y:= ptr↑.field;        need residency check here
 ...
 ptr↑.field := x;       check redundant if ptr unchanged
end;
```

Under our idempotency assumption, so long as *ptr* is not updated, the second time *ptr* is dereferenced a residency check is superfluous.

To optimize away redundant residency checks we perform data flow analysis similar to common sub-expression elimination (CSE). However, unlike CSE, which saves the value of a computation for use in a later expression that makes use of the same computation, residency checks have a side effect, turning the checked reference into a pointer to an in-memory object. Thus, residency check elimination requires the addition of a new optimization.

In the presence of incremental buffer removal we can no longer assume the idempotency of residency checks, since a data item may become non-resident at any time. For this reason we must maintain a contract between the compiler and the run-time system, so that any residency assumptions made by the compiler will be maintained by the buffer manager. We could simply disallow run-time removal of buffers; this is likely to be unacceptable. Instead, we arrange to have residency assumptions re-established after a buffer is removed. One way to do this is to have the buffer manager scan the stack to re-fault any object that was assumed resident. To support this, the compiler must generate information about the residency assumptions for each stack allocated variable at all points in the program at which a buffer may be removed. Alternatively, the buffer manager can modify the return addresses of each stack frame so that when a frame is returned to it first re-establishes its residency assumptions before continuing. Once again, the compiler must support this by generating a code fragment to re-establish the residency assumptions for each potential buffer removal point in the program, along with the information needed to patch the return address.

4.3 Co-residency

Object faulting treats the persistent store as a directed graph: the nodes of the graph are the objects, while the edges in 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. Let us call the objects referred to by an object its *children*, and the object itself the *parent*.[4] We partition the children of an object into two categories: those that are co-resident with the parent, and those that are not. Whenever we fault the parent, each pointer to a co-resident child is swizzled to point directly at the child, instead of a fault block for the child. If the child is not yet resident it must also be faulted. The compiler can then omit any residency check where a pointer from a parent to its co-resident child is dereferenced, since the pointer is guaranteed to refer directly to the memory-resident child.

The question remains as to how such co-residency information can be presented to the compiler. The *type graph* of a program is an abstract and static representation of the potential graph structure of the persistent store. Its nodes are the types of the program. If an object of type T contains a pointer to an object of type U then there will be an edge from T to U in the type graph. We annotate the edges of the type graph to represent co-residency constraints. One way to do this is to annotate the types of a program as in the following Modula-3 code:

```
type
 RT = record
        field: cores persistent ref T;
        val: integer;
      end;
var
 p: persistent ref RT:= ...;
 t: T;
begin
 p↑.val := 4;        faults both p↑ and p↑.field↑
 ...
 t := p↑.field↑;     no residency check needed since p↑.field↑ already resident
 ...
end;
```

Here, dereferencing the *field* pointer of the record does not need a residency check, since the item it refers to has already been made resident.

Co-residency declarations must be considered an integral part of a type's definition. Since Modula-3 uses structural type equivalence to determine if two types are the same, this means types having different co-residency declarations cannot be equivalent. While it is tempting to consider

[4]This terminology is not intended to reflect the existence of any hierarchy of objects, but rather the directedness of the pointer relationship.

relaxing this requirement, without it co-residency declarations will not be useful to the compiler. Suppose that T and U have different co-residency declarations, but are otherwise structurally equivalent. If we consider them to be the same type then a variable of type **persistent ref** T can actually be used to refer to an instance of type U at run time. This instance will not have the same co-residency declarations enforced on it as an instance of T, so the compiler cannot safely eliminate residency checks based on the declared type of the variable, rendering the co-residency constraints useless. For the same reason, a subtype must inherit its supertype's co-residency declarations.

Recursively defined types form cycles in the type graph. We treat co-residency declarations that form a cycle as deliberate indications that the entire recursively defined structure is to be made co-resident. Consider a type definition for the nodes of a binary tree:

```
type binaryTreeNode = record
    value: integer;
    left:  cores persistent ref binaryTreeNode;
    right: persistent ref binaryTreeNode;
  end;
```

Whenever a node of the tree is made resident, the entire left spine of the subtree rooted at that node will also be made resident. Similarly, we treat co-residency declarations on pointer elements of an open array type as indicating that every item referred to by the array is to be made resident.

Finally, even though our examples have annotated the type declarations in the code, it is perhaps preferable for co-residency declarations to be maintained by some tool, in files auxiliary to the code. An ill-placed co-residency constraint will retrieve more objects than are not needed, and so will be more damaging to performance than if it was omitted. Thus, it is important that good co-residency declarations be derived. This requires an understanding of both the dynamic behavior of the programs using those declarations and the physical characteristics of the persistent store. Static analysis can provide some information about the dynamic behavior of a program. This analysis can be supplemented by statistics gathered at run time. The compiler can support such profiling by inserting statistics-gathering code at interesting points in the program at compile time. We envision the analysis and profiling of whole suites of programs, so that a global perspective can be obtained by which the combined behavior of all the programs acting on the persistent store may be better understood.

4.4 Clustering

We have mentioned that storage managers may place objects physically close together on disk, so that they may all be retrieved with one disk access. This is known as *clustering*. A *good* clustering will group objects that have a high probability of being accessed at the same time. Note

that we do not mean clustering for virtual memory, where the unit of clustering is a fixed size page; our clusters may comprise multiple pages.

We have used co-residency information to indicate objects that must be made resident at the same time, so that their retrieval is triggered by just one residency check. If these objects can be clustered, then the retrieval itself will be performed with fewer disk accesses. In this way we can see co-residency as driving clustering decisions: when new data items are made persistent their co-residency constraints are a useful basis on which to cluster them.

The programs that manipulate the store influence its clustering decisions. However, recompiling a program is less expensive than re-clustering a large persistent store, so we should consider turning the dependency around. After adapting a program's co-residency constraints to match the prevailing clusters in the store, the program can be recompiled with those new constraints. Like co-residency, clustering decisions can be made based on edge traversal statistics gathered with the support of the compiler.

It is important to realize that clustering is likely to have a bigger impact on performance than any of the other optimizations we have described, since the performance gains of CPU-time optimizations are likely to be dominated by those of I/O optimizations (most likely by an order of magnitude).

4.5 Swizzling

We can use the compiler to support efficient swizzling and unswizzling. For each type in a program the compiler generates a pair of routines to perform the swizzling and unswizzling operations for instances of the type (these can be implemented as hidden methods). When an object is first faulted it will be swizzled by a call to the swizzling routine of its type. Likewise, when it is to be removed from memory it will be unswizzled by a call to its unswizzling routine. To meet the co-residency constraints of an object, its swizzling routine will call the swizzling routines of any co-resident children. Similarly, an object's unswizzling routine will call the unswizzling routines of its co-resident children, making placement requests on the storage manager so that newly created co-resident children are clustered with their parents.

The alternative to type-specific compiler-generated swizzling routines is an interpreted approach: a general-purpose routine swizzles each object based on descriptor information stored with its type code. Clearly, this will suffer from the overhead of having to decode the descriptor for each object. In the compiled approach this overhead is eliminated in favor of fast, straight-line, compiled code.

4.6 Evaluation

We can characterize the behavior of many persistent programs as a traversal of a persistent directed graph, starting from a known root node, and performing some work at each node visited. Each node of the graph is stored as a single object in the persistent store. We would expect redundant residency check elimination to reduce the number of residency checks to 1 check per node visit.

If a strictly object-oriented programming style is observed we would expect the number of residency checks to be reduced to less than 1 per node visit, since the implementation for method invocation on a persistent object (as described in Section 3.2) folds the residency check on the object into the dynamic binding of the call. Of course, any gain here is offset by the dynamic nature of the call binding mechanism. In the case that a particular method invocation can be bound statically, or if it is inlined, residency checks will need to be inserted in the method code, so that the figure will be closer to 1 check per node visit.

We can expect co-residency information to reduce the number of residency checks to 1 check per pointer variable, so long as only co-resident objects are accessed from that variable. Even if we only visit the co-resident children of an object once, we still expect the retrieval style that co-residency allows—where a group of co-resident objects are retrieved together at the cost of one residency check—to result in some overall improvement.

These observations give us an insight into the effectiveness of compile-time optimizations for persistent programs. Our goal is to undertake a quantitative performance evaluation of compiler support for persistence. Not only will this better allow us to evaluate the optimizations we have described, we will also be able to obtain an appreciation for the gains that compilation provides for swizzling, unswizzling, and profiling.

5 Related Work

The only other work of which we are aware on optimizing persistent programs at compile time is that undertaken by Richardson in his database programming language E [Richardson, 1989]. He performs an optimization much like the redundant residency check elimination we have described. However, the other optimizations we have described are not performed in E. Importantly, an E object is always accessed via a level of indirection. Our object faulting techniques and co-residency optimization eliminate such indirection.

Some progress has been made towards understanding clustering issues in persistent object stores [Benzaken, 1990; Benzaken and Delobel, 1990; Shannon and Snodgrass, 1990]. However, none of this work considers using clustering information at compile time to generate programs that run efficiently against the persistent store.

There is a large body of background work on persistent programming languages, database programming languages, and persistent object stores. Published accounts of this work may be found in the relevant journals, conferences, and workshops.

6 Conclusions

We have devised an implementation approach to persistence that allows the compiler to significantly improve the performance of persistent programs. In so doing we have framed the performance problem for persistent programs in terms of two cost components: CPU-time costs and I/O costs. We have identified a number of optimizations that may be applied to help reduce these costs, and have made intuitive arguments as to the performance enhancements to be gained by them. We look forward to confirming these intuitions with quantitative performance evaluation.

References

[Atkinson *et al.*, 1983] 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 (Nov. 1983), 360–365.

[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.

[Benzaken, 1990] Véronique Benzaken. An evaluation model for clustering strategies in the O_2 object-oriented database system. Tech. Rep. 49-90, Altaïr, BP105, 78153 Le Chesnay Cedex, France, Aug. 1990.

[Benzaken and Delobel, 1990] Véronique Benzaken and Claude Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In [Dearle *et al.*, 1990].

[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 Data Bases* (Kyoto, Japan, Aug. 1986), Morgan Kaufmann, 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 Frederick H. Lochovsky, Eds. ACM Press/Addison-Wesley, New York, New York, 1989, ch. 14, pp. 341–369.

[Dearle *et al.*, 1990] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, Eds. *Implementing Persistent Object Bases: Principles and Practice.* Fourth International Workshop on Persistent Object Systems: Design, Implementation, and Use (Martha's Vineyard, Massachusetts, Sept. 1990), Morgan Kaufmann, 1990.

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

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

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

[Moss, 1990b] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. COINS Technical Report 90-38, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, May 1990. Submitted for publication.

[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.

[Richardson and Carey, 1987] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementations in EXODUS. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, May 1987), *ACM SIGMOD Rec. 16*, 3 (Dec. 1987), pp. 208–219.

[Shannon and Snodgrass, 1990] Karen Shannon and Richard Snodgrass. Semantic clustering. In [Dearle *et al.*, 1990].

[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, Pennsylvania, Apr. 1984), *ACM SIGPLAN Not. 19*, 5 (May 1984), pp. 157–167.

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