# Fine-Grained Mobility in the Emerald System

Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black

Department of Computer Science
University of Washington
Seattle, WA 98195

**Abstract**

Emerald is an object-based language and system designed for the construction of distributed programs. An explicit goal of Emerald is support for object mobility; objects in Emerald can freely move within the system to take advantage of distribution and dynamically changing environments. We say that Emerald has *fine-grained* mobility because Emerald objects can be small data objects as well as process objects. Fine-grained mobility allows us to apply mobility in new ways but presents implementation problems as well. This paper discusses the benefits of fine-grained mobility, the Emerald language and run-time mechanisms that support mobility, and techniques for implementing mobility that do not degrade the performance of local operations. Performance measurements of the current implementation are included.

## 1 Introduction

Process migration has been implemented or described as a goal of several distributed systems [19, 13, 15, 18, 7, 23, 8]. In these systems, entire address spaces are moved from node to node. For example, a process manager might initiate a move to share processor load more evenly, or users might initiate remote execution explicitly. In either case, the running process is typically ignorant of its location and unaffected by the move.

During the last three years, we have designed and implemented Emerald [4, 5], a distributed object-based language and system. A principal goal of Emerald is to experiment with the use of mobility in distributed programming. Mobility in the Emerald system differs from existing process migration schemes in two important respects. First, Emerald is object-based and the unit of distribution and mobility is the object. While some Emerald objects contain processes, others contain only data: arrays, records, and single integers are all objects. Thus, the unit of mobility can be much smaller than in process migration systems. Object mobility in Emerald subsumes both process migration and data transfer. Second, Emerald has language support for mobility. Not only does the Emerald language explicitly recognize the notions of location and mobility, but the design of conventional parts of the language (e.g., parameter passing) is affected by mobility.

This paper appeared in *ACM Trans. on Computer Systems* 6(1), February 1988; it was presented at the 11th ACM Symp. on Operating Systems Principles, December 1987. This work was supported in part by the National Science Foundation under Grants No. MCS-8004111 and DCR-8420945, by Københavns Universitet (the University of Copenhagen), Denmark under Grant J.nr. 574-2,2, by a Digital Equipment Corporation External Research Grant, and by an IBM Graduate Fellowship.

The advantages of process migration, which have been noted in previous work, include:

1. *Load sharing* – By moving objects around the system, one can take advantage of lightly used processors.

2. *Communications performance* – Active objects that interact intensively can be moved to the same node to reduce the communications cost for the duration of their interaction.

3. *Availability* – Objects can be moved to different nodes to provide better failure coverage.

4. *Reconfiguration* – Objects can be moved following either a failure or a recovery, or prior to scheduled down-time.

5. *Utilizing special capabilities* – An object can move to take advantage of unique hardware or software capabilities on a particular node.

Along with these advantages, fine-grained mobility provides three additional benefits:

1. *Data Movement* – Mobility provides a simple way for the programmer to move data from node to node without having to explicitly package data. No separate message passing or file transfer mechanism is required.

2. *Invocation Performance* – Mobility has the potential to improve the performance of remote invocation by moving parameter objects to the remote site for the duration of the invocation.

3. *Garbage Collection* – Mobility can help simplify distributed garbage collection by moving objects to sites where references exist [13, 24].

To our knowledge, the only other system that implements object mobility in a style similar to Emerald is a recent implementation of distributed Smalltalk [3].

In addition to mobility and distribution, Emerald is intended to provide efficient execution. We wanted to achieve performance competitive with standard procedural languages in the local case and standard remote procedure call systems in the remote case. These goals are not trivial in a location-independent object-based environment. To meet them, we have relied heavily on an appropriate choice of language semantics, a tight coupling between the compiler and run-time kernel, and careful attention to implementation.

Emerald is not intended to run in large, long-haul networks. We assume a local area network with a modest number of nodes (e.g., 100). In addition, we assume that nodes are homogeneous in the sense that they all run the same instruction set, and that they are trusted.

In this paper we concentrate primarily on the language and run-time mechanisms that support fine-grained mobility while retaining efficient intra-node operation. First, we present a brief overview of the Emerald language and system, and its mobility and location primitives. A more detailed description of object structure in Emerald can be found in [4], and of the type system in [5]. Second, we discuss the implementation of fine-grained mobility in Emerald and new problems that arise from providing such support. Third, we present measurements of the implementation and draw implications from the measurements and our design experience.

## 2 Overview of Emerald

As previously stated, an important goal of Emerald was explicit support for mobility. From a conceptual viewpoint, a more important goal was a single object model. Object-based systems typically lie at the ends of a spectrum: object-based languages such as Smalltalk [10] and CLU [17] provide small, local, data objects; object-based operating systems, like Hydra [25] and Clouds [1], provide large, active objects. Distributed systems such as Argus [16] and Eden [2] that support both kinds of object have a separate object definition mechanism for each. Choosing the right mechanism requires that the programmer know ahead of time all uses to which an object will be put; the alternative is to accept the inefficiency and inconvenience of using the "wrong" mechanism, or to reprogram the object later as needs change. For example, while programming a Collaborative Editing System in Argus, Greif et al. have observed that a designer can be forced to use a Guardian where a cluster might be more appropriate [11].

The motivation for two distinct definition mechanisms is the need for two distinct implementations. In distributed object-based systems such as Clouds and Eden, a *local* execution of the general invocation mechanism can take milliseconds or tens of milliseconds [21]. A more restrictive and efficient implementation is appropriate for objects that are known to be always local; for example, shared store can be used in preference to messages.

While we believe in the importance of multiple implementations, we do not believe that these need to be visible to the programmer. In Emerald, programmers use a single object definition mechanism with a single semantics for defining all objects. This includes small, local, data-only objects and active, mobile, distributed objects. However, the Emerald compiler is capable of analyzing the needs of each object and generating an appropriate implementation. For example, an array object whose use is entirely local to another object will be implemented differently from an array that is shared globally. The compiler produces different implementations from the same piece of code, depending on the context in which it is compiled [14].

The motivation for designing a new language, rather than applying these ideas to an existing language, is that the semantics of a language often preclude efficient implementation in either the local or remote case. In designing Emerald, we kept both implementations in mind. Moreover, Emerald's unique type system allows the programmer to state either nothing or a great deal about the use of a variable; in general, the more information the compiler has, the better the code that it generates.

We believe that the current Emerald implementation demonstrates the viability of this approach and meets our goal of local performance commensurate with procedural languages. Table 1 shows the performance of several local Emerald operations executed on a MicroVAX II[1]; more details on the compiler and its implementation can be found in [14]. The "resident global invocation" time is for a global object (i.e., one that can move around the network) when invoked by another object resident on the same node.

For comparison with procedural languages, a C procedure call takes 13.4 microseconds, while a Concurrent Euclid procedure call takes 16.4 microseconds. Concurrent Euclid is slower because, like Emerald, it must make explicit stack overflow checks on each call.

[1]MicroVAX is a trademark of Digital Equipment Corporation.

| Emerald Operation | Example | Time/$\mu$s |
|---|---|---|
| primitive integer invocation | $i \leftarrow i + 23$ | 0.4 |
| primitive real invocation | $x \leftarrow x + 23.0$ | 3.4 |
| local invocation | localobject.no-op | 16.6 |
| resident global invocation | globalobject.no-op | 19.4 |

Table 1: Timings of Local Emerald Invocations

### 2.1 Emerald Objects

Each Emerald object has four components:

1. A unique network-wide name.
2. A representation, i.e., the data local to the object, which consists of primitive data and references to other objects.
3. A set of operations that can be invoked on the object.
4. An optional process.

Emerald objects that contain a process are active; objects without a process are passive data structures. Objects with processes make invocations on other objects, which in turn invoke other objects, and so on to any depth. As a consequence, a thread of control originating in one object may span other objects, both locally and on remote machines. Multiple threads of control may be active concurrently within a single object; synchronization is provided by monitors.

Figure 1 shows an example definition of an Emerald object – in this case a simple directory object called *aDirectory*. The representation of the object consists of an array, *a*, of directory elements. The object exports three operations: Add, Lookup, and Delete. The array *a* and the operations are defined within a monitor to guarantee exclusive access to the array.

### 2.2 Types in Emerald

The Emerald language supports the concept of *abstract type* [5]. The abstract type of an object defines its interface: the number of operations that it exports, their names, and the number and abstract types of the parameters to each operation. For example, consider the abstract type definition for *SimpleDirectoryType* below:

**const** *SimpleDirectoryType* == **type** *SimpleDirectoryType*
  **operation** *Lookup*[ **String** ] → [ **Any** ]
  **operation** *Add*[ **String**, **Any** ]
**end** *SimpleDirectoryType*

This defines an abstract type with two operations, *Lookup* and *Add*. *Lookup* has an input parameter of abstract type **String** and returns an object of abstract type **Any**. We say that an object *conforms* to an abstract type if it implements at least the operations of that abstract

```
object aDirectory
  export Add, Lookup, Delete
  monitor
    const DirectoryElement == record DirectoryElement
      var name : String
      var obj : Any
    end DirectoryElement

    const a == Array.of[DirectoryElement].empty

    function Lookup[n : String] → [o : Any]
      var element : DirectoryElement
      var i : Integer ← a.lowerbound
      loop
        exit when i > a.upperbound
        element ← a.getelement[i]
        if element.getname = n then
          o ← element.getobj
          return
        end if
        i ← i + 1
      end loop
      o ← nil
    end Lookup

    % Implementation of Add and Delete
  end monitor
end aDirectory
```

Figure 1: An Emerald Directory Object Definition

type, and if the abstract types of the parameters conform in the proper way. When an object is assigned to a variable, the abstract type of that object must conform to the declared abstract type of the variable. All objects conform to type **Any**, since **Any** has no operation.

Abstract types permit new implementations of an object to be added to an executing system. To use a new object in place of another, the abstract type of the new object must conform to the required abstract type. For example, we could assign the object *aDirectory* in Figure 1 to a variable declared to have abstract type *SimpleDirectoryType* because *aDirectory* conforms to *SimpleDirectoryType*. Note that each object can implement a number of different abstract types, and an abstract type can be implemented by a number of different objects.

Emerald has no class/instance hierarchy, in contrast to Smalltalk. Objects are not members of a class; conceptually, each object carries its own code. This distinction is important in a distributed environment where separating an object from its code would be costly. However, identically implemented Emerald objects on each node do share code. In the implementation, the code is stored in a *concrete type object*. Because concrete type objects are immutable, they can be freely copied. When an object is moved to another node, only its data is moved. If the object contains a process, part of that data will include the process' stack, but no code is transferred.

When a kernel receives an object, it determines whether a copy of the concrete type object implementing that object already exists locally; if it does not, the kernel obtains a copy of it by finding one on another node using the location algorithm (described in Section 3.2). Typically the concrete type will be available from the node that sent the object. When a concrete type object arrives, it is dynamically linked into the kernel – the compiler generates relocatable code and sufficient symbol table information to make such dynamic linking possible. This scheme makes it possible to dynamically add new concrete types that implement existing abstract types. Concrete type objects are kept on a node for as long as there are objects referencing them, after which they are garbage collected.

## 2.3 Primitives for Mobility

Object mobility in Emerald is provided by a small set of language primitives. An Emerald object can:

- *Locate* an object (e.g., "**locate** X" returns the node where X resides).
- *Move* an object to another node (e.g., "**move** X **to** Y" co-locates X with Y).
- *Fix* an object at a particular node (e.g., "**fix** X **at** Y").
- *Unfix* an object, making it mobile again following a fix (e.g., "**unfix** X").
- *Refix* an object, atomically performing an Unfix, Move, and Fix at a new node (e.g., "**refix** X **at** Z").

The move primitive is actually a hint; the kernel is not obliged to perform the move and the object is not obliged to remain at the destination site. Fix and refix have stronger semantics; if the primitives succeed the object will stay at the destination until it is explicitly unfixed.

Central to these primitives is the concept of location, which is encapsulated in a *node* object. A node object is an abstraction of a physical machine. Location may be specified by naming either a node object or any other object. If the programmer specifies a non-node object, the location implied is the node on which that object resides. These concepts are similar to the location dependent primitives in Eden [6].

A crucial issue when moving objects containing references is deciding how much to move [20]. An object is part of a graph of references, and one could move a single object, several levels of objects, or the entire graph. The simplest approach — moving the specified object alone — may be inappropriate. Depending on how the object is implemented, invocations of the moved object may require remote references that would have been avoided if other related objects had been moved as well.

The Emerald programmer may wish to explicitly specify which objects move together. For this purpose, the Emerald language allows the programmer to *attach* objects to other objects. When a variable is declared, the programmer can specify the variable to be an "attached variable".

For example, in the Emerald mail system, mail messages have four fields: a sender, an array of destination mailboxes, a subject line and a text string. It makes sense for the array of destination mailboxes to be attached to the mail message, and this could be specified as:

**attached var** *ToList* : **Array**.*of*[ *Mailbox* ]

When the mail message is moved, the array pointed to at that time by *ToList* is moved with it. This may effect the performance of invocations on *ToList*, but not their semantics.

Attachment is transitive: any object attached to *a* will also be moved. For example, linked structures may be moved as a whole by attaching the link fields. Attachment is not symmetric; the object named by *a* can itself be moved, perhaps before it is invoked, and no attempt will be made to move *directory* with it.

## 2.4 Parameter Passing

An important issue in the design of distributed, object-based systems (as well as remote procedure call systems) is the choice of parameter passing semantics. In an object-based system, all variables refer to other objects. The natural parameter passing method is therefore call-by-object-reference, where a reference to the argument object is passed. This is in fact the semantics chosen by CLU (where it is called *call by sharing*) [17] and Smalltalk [10].

In a distributed object-oriented system, the desire to treat local and remote operations identically leads one to use the same semantics. However, such a choice could cause serious performance problems: on a remote invocation, access by the remote operation to an argument is likely to cause an additional remote invocation. For this reason, systems such as Argus have required that arguments to remote calls be passed by value, not by object-reference [12]. Similarly, remote procedure call systems require call-by-value since addresses are context-dependent and have no meaning in the remote environment.

The Emerald language uses call-by-object-reference parameter passing semantics for all invocations, local or remote. In both cases, the invoking code constructs an activation record that contains references to the argument objects. In the local case, the invoked object is called directly and receives a pointer to the activation record for the invocation. In the remote case, the activation record must be reconstructed on the remote system, but the basic operation and semantics are identical.

Because Emerald objects are mobile, it may be possible to avoid many remote references by moving argument objects to the site of a remote invocation. Whether this is worthwhile depends on (1) the size of an argument object, (2) other current or future invocations of the argument, (3) the number of invocations that will be issued by the remote object to the argument, and (4) the relative costs of mobility and local and remote invocation.

In the current Emerald prototype, arguments are moved in two cases. First, based on compile-time information, the Emerald compiler may decide to move an object along with an invocation. For example, small immutable objects are obvious candidates for moving because they can be copied cheaply. Obviously, it makes little sense to send a remote reference to a small string or integer. Second, the Emerald programmer may decide that an object should be moved based on knowledge about the application. To make this possible, Emerald provides a parameter passing mode that we call *call-by-move*. Call-by-move does not change the semantics, which is still call-by-object-reference, but at invocation time the argument object is relocated to the destination site. Following the call the argument object may either return to the source of the call or remain at the destination site (we call these two modes *call-by-visit* and *call-by-move*, respectively).

Call-by-move is a convenience and a performance optimization. Arguments could instead be moved by explicit move statements. However, providing call-by-move as a parameter passing mode allows packaging of the argument objects in the same network packet as the invocation message.

As an example, consider another mail system example. After composing a mail message (whose fields were described previously), the user invokes the message's *Deliver* operation:

```
operation Deliver
   var aMailbox : Mailbox
   if ToList.length = 1 then
      aMailbox ← Tolist.getelement[ ToList.lowerbound ]
      aMailbox.Deliver[ move self ]
   else
      var i : Integer ← ToList.lowerbound
      loop
         exit when i > ToList.upperbound
         aMailbox ← ToList.getelement[ i ]
         aMailbox.Deliver[ self ]
         i ← i + 1
      end loop
   end if
end Deliver
```

This operation delivers the message to all the mailboxes on the *ToList*. However, in the common case where there is only one destination, call-by-move is used to co-locate the mail message with the (single) destination mailbox.

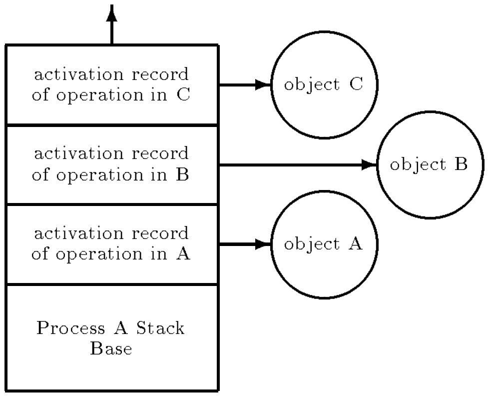


Figure 2: Process Stack and Activation Records

### 2.5 Processes, Objects, and Mobility

An Emerald process is a thread of control that is initiated when an object with a process is created. A process can invoke operations on its object or on any object that it can reference. We think of a process as being a stack of activation records, as shown in Figure 2. The thread of control of one object's process may pass through other objects; in the case of Figure 2, the process owned by object $A$ invokes operations in objects $A$, $B$, and $C$.

One can think of remote invocations in several ways. In the traditional remote operation model [22], the sending process blocks and an existing remote process executes the operation, possibly returning a value to the caller, which then continues execution. In Emerald, when a remote invocation occurs, we think of the process moving to the destination node and invoking the object there. Or, alternatively, the new activation record moves to the destination node to become the base of a new segment of the process stack on that node. The invocation stack of a single Emerald process can therefore be distributed across several nodes.

Mobility presents a special problem to this process structure. For example, given the process activation stack in Figure 2, suppose that object $B$ is moved to another node. In that case, the part of the thread that is executing in $B$ must move along with $B$; that is, the activation record must move. Furthermore, when the operation in $C$ terminates it must now return to a different node. If object $C$ were to move to a different node from $B$, we would have three parts of the process stack on three different nodes. Invocation returns would propagate control back from node to node.

One could imagine a different scheme that left the stack intact, with invocations always returning to the node on which the root process resides; at that point the situation could be analyzed and control passed to the proper location. The problem with this design is that it leaves *residual dependencies*. In the situation where objects $C$ and $B$ have moved to different nodes, it should be possible for control to return from $C$ to $B$ even if $A$ is temporarily unreachable. Depending on $B$'s behavior, it may in fact be some time before a return to $A$ or its node is actually required. Moving invocation frames along with the objects in which they execute ensures that execution can continue as long as possible, and removes the computational burden from nodes that do not need to be involved in a communication.

# 3 Implementing Mobility in Emerald

Adding process mobility to existing systems often proves to be a difficult task. One problem is extracting the entire state of a process, which may be distributed through numerous operating system data structures. Second, the process may have variables that directly index those operating system data structures, such as open file descriptors, window numbers, etc.

In a distributed object-based system, this problem may be somewhat simplified. Objects cleanly define the boundaries of all system entities. Furthermore, since all resources are objects, addressing is standardized and location-independent. All objects, whether user-implemented or kernel-implemented, are addressed indirectly using an object ID. Operations are performed through a standard invocation interface.

While distribution and mobility increase the generality of a system they often reduce its performance. Anyone building an object-based system must be sensitive to performance because of the generally poor performance of such systems. The implementation of mobility in Emerald involved tradeoffs between the performance of mobility and that of more fundamental mechanisms, such as local invocation. Where possible, we have made these tradeoffs in favor of the performance of frequent operations, and we would typically be willing to increase the complexity of mobility to save a microsecond or two on local invocation. Furthermore, it takes a hundred times longer to move an object than to perform a local invocation; adding 5 microseconds to the object move time makes little relative difference, while 5 microseconds is 25 percent of the local invocation time. The result of this philosophy is that, to a great extent, the existence of mobility and distribution in Emerald do not interfere with the performance of objects on a single node.

In the following sections, we describe some of the implementation of the Emerald kernel that is relevant to mobility, and some of the tradeoffs that we have made in this design.

## 3.1 Object Implementation and Addressing

To meet our goal of building a distributed object-based system with efficient local execution, the Emerald implementation relies heavily on shared memory. We have implemented a prototype of Emerald on top of DEC's Ultrix system (which is based on Unix 4.2BSD) running on five DEC MicroVAX II workstations[2]. The Emerald kernel and all Emerald objects on a single node execute within a single Ultrix address space. Emerald processes are lightweight threads scheduled within that address space. Protection among objects is guaranteed by the compiler both through type checking and through run-time checks inserted into the code. Objects that are resident on

[2]Unix is a trademark of AT&T. Ultrix is a trademark of Digital Equipment Corporation.

the same node address each other directly — an implementation style that has implications for mobility.

As previously stated, all objects are coded using a single object definition mechanism. However, based on its knowledge of an object's use, the compiler is free to choose an appropriate addressing mechanism, storage strategy, and invocation protocol [14]. The Emerald compiler uses three different styles of object implementation:

- A *global* object can be moved independently, can be referenced globally in the network, and can be invoked by objects not known at compile time. Global objects are heap allocated. An invocation of such an object may require a remote invocation. In Figure 1, the object *aDirectory* is implemented as a global object.

- A *local* object is completely contained within another object; that is, a reference to the local object is never exported outside the boundary of the enclosing object. Such objects cannot move independently; they always move along with their enclosing object. Local objects are heap allocated. An invocation is implemented by a local procedure call or inline code. The array $a$ in Figure 1 is not used outside of the directory and can thus be implemented as a local object.

- A *direct* object is a local object whose data area is allocated directly in the representation of the enclosing object. Direct objects are used mainly for primitive built-in types, structures of primitive types, and other simple objects whose organization can be deduced at compile time. For example, all integers are direct objects.

Figure 3 shows the various implementation and addressing options used by Emerald. Variable $X$ names a global object and the value stored in $X$ is the address of a local *object descriptor*. Each node contains an object descriptor for every global object for which references exist on that node. When the last reference to object $m$ is deleted from node $k$, $k$'s object descriptor for $m$ can be garbage collected.

An object descriptor contains information about the state and location of a global object. The first word of the object descriptor identifies it as a descriptor and contains control bits indicating whether the object is local or global (the G bit) and whether or not the object is resident (the R bit). If the resident bit is set, the object descriptor contains the memory address of the object's data area; otherwise, the descriptor contains a forwarding address to the object as described in Section 3.2.

Variable $Y$ in Figure 3 names a local object. The value stored in $Y$ is the address of the object's data area. The first word of this data area, like the first word of an object descriptor, contains fields identifying the area and indicating that this is a local object, i.e., the data area acts as its own descriptor. Finally, variable $Z$ names a direct object that was allocated within the variable itself.

Notice that within a single node, all objects can be addressed directly without kernel intervention. Emerald variables contain references that are location *dependent*, that is, they have meaning only within the context of a particular node. For invocation of global objects, compiled code first checks the *resident* bit to see if a local invocation can be performed directly. If the target object is not resident, the compiled code will trap to the kernel so that a remote invocation

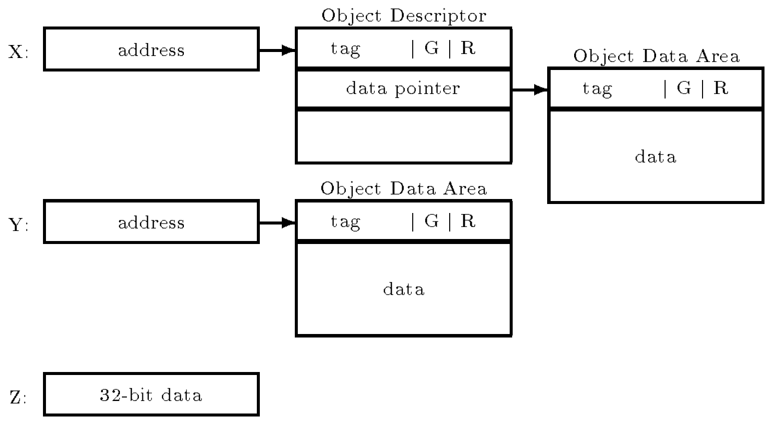


Figure 3: Emerald Addressing Structures

can be performed. In this way, global objects can be invoked locally in time comparable to a local procedure call.

## 3.2 Finding Objects

Since objects are allowed to move freely, it is not always possible to know the location of a given object, e.g., when invoking it. The run-time system must keep track of objects or at least be able to find them when needed. Keeping every node in the system up-to-date on the current location of every object is expensive and unnecessary. Instead, we use a scheme based on the concept of *forwarding addresses* as described by Fowler [9].

Each global object is assigned a unique network-wide *Object Identifier (OID)*, and each node has a hashed *access table* mapping OIDs to object descriptors. The access table contains an entry for each local object for which a remote reference exists and each remote object for which a local reference exists.

As previously described, an object descriptor contains a *forwarding address* as well as the object's OID. A forwarding address is a tuple $<$ *timestamp, node* $>$ where the node is the last known location of the object and the timestamp specifies the age of the forwarding address. Fowler [9] has shown that it is sufficient to maintain the timestamp as a counter incremented every time the object moves. Given conflicting forwarding addresses for the same object, it is simple to determine which one is most recent. Every reference sent across a node boundary contains the OID of the referenced object and the latest available forwarding address. The receiving node may then update its forwarding address for the referenced object, if required.

If an object is moved from node $A$ to node $B$, both $A$ and $B$ will update their forwarding addresses for the object. No action is taken to inform other nodes. Should node $C$ try to invoke the object at $A$, $A$ will forward the invocation message to $B$. When the invocation completes, $B$ will send the reply to $C$ with the new forwarding address piggybacked onto the reply message.

An alternative strategy, which we did not adopt, would be to keep track of all nodes that have references to a particular object. Should that object move, update messages could be sent to those nodes. However, these extra messages could significantly increase the cost of move and of passing references. For example, when an object reference is passed to a node for the first time, that node would have to register with the node responsible for the object. The DEMOS/MP system used a forwarding address update scheme, and updating forwarding addresses was shown to incur significant overhead [18]. In addition, sending update messages on every move will not avoid the need for invocation forwarding, since update messages do not arrive immediately at all destinations. Our scheme places the cost of forwarding address maintenance on the current users of a forwarding address.

When it is necessary to locate an object, for example when the *locate* primitive is used, we apply the following algorithm. If the kernel has a forwarding address for the object, it asks the specified node whether the object is resident there; if it is we are done. Otherwise, if that node has a newer forwarding address, then we start over using that forwarding address. However, if that node is unreachable or has no better information, we resort to a broadcast protocol.

The broadcast protocol is used whenever the previous step has failed to find the object. The searching kernel sends a first broadcast message to all other nodes seeking the location of the object. To reduce message traffic, only a kernel that has the specified object responds to the broadcast. If the searching kernel receives no response within a timelimit, it sends a second broadcast, this time requesting a positive or negative reply from all other nodes. All nodes not responding within a short time are sent a reliable, point-to-point message with the location request. If every node responds negatively we conclude that the object is unavailable.

When performing remote invocations, the invocation message is sent without locating the target object first. Only if there is a lost forwarding address somewhere along the path will the location algorithm be used. This optimizes for the common case where the object has not moved or where a valid forwarding address exists.

### 3.3 Finding and Translating Pointers

The use of direct memory addresses in Emerald (as opposed to indirect references, such as those used in the standard Smalltalk implementation [10]) increases the performance of local invocations. Consequently, movement of an object involves finding and modifying all of the direct addresses, increasing the cost of mobility. We feel that this is reasonable, since motion is less frequent than invocation. This design places the price of mobility on those who use it.

Finding and translating references could be done in several ways. For example, a *tag* bit in each word could indicate whether or not the word contains an object reference. Smalltalk 80 uses such bits to distinguish integers from references, but using tags increases the overhead of arithmetic operations and complicates the implementation in general.

Instead, the Emerald compiler generates *templates* for object data areas describing the layout of the area. The template is stored with the code in the concrete type object that defines the object's operations. Each object data area contains a reference to the concrete type object so

```
const simpleobject == object simpleobject
  monitor
    var myself : Any ← simpleobject
    var name : String ← "Emerald"
    var i : Integer ← 17
    operation GetMyName → [n : String]
      n ← name
    end GetMyName
          ⋮
  end monitor
end simpleobject
```

Figure 4: Simple Emerald Object Definition

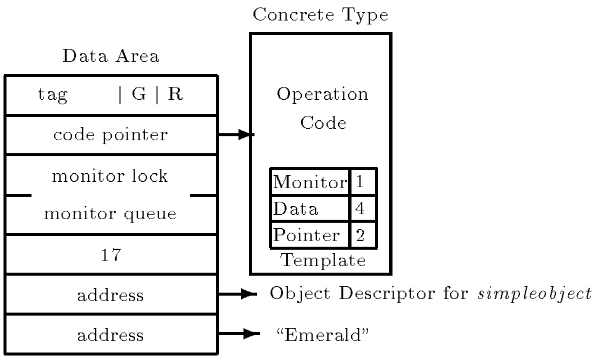


Figure 5: Data Area and Template Structure

that the code and the template can be found given only the data area. In addition to their use for mobility, templates are used for garbage collection and debugging, as these tasks must also understand an object's data area.

As an example, consider the Emerald program shown in Figure 4 which defines a single object containing three variables inside a monitor. The variable *myself* contains a pointer to its own object descriptor. The variable *name* is initialized to point to a local string object. The variable *i* does not contain a pointer since integers are implemented as direct objects. The corresponding object data area and template are shown in Figure 5.

The data area for *simpleobject* contains:

- control information as described earlier.
- a pointer to the code for *simpleobject*.
- a lock for the monitor.

- the variable $i$ allocated as 4 bytes of data.
- the variables *myself* and *name*, each allocated as a pointer to an object.

The template does not describe the first two items since every data area contains them. Each template entry contains a count of the number of items described and the types of the items (called *template-types*). Typical template-types are:

- *Pointer*, which is the address of an object; pointers must be translated if the object is moved.
- *Data*, which is direct data (e.g., integers) stored as a number of bytes; these are not translated.
- *MonitorLock*, which controls access to the object's monitor. Monitors are implemented as a Boolean and a queue of processes awaiting entry to the monitor. A monitor must be translated if the object is moved.

Attached objects, which must move along with an object being moved, are indicated simply by a bit in the template entry. The compiler contiguously allocates variables that can be described by identical template entries. Therefore, the average template contains only two or three entries.

In addition to data areas, the compiler must produce templates to describe activation records so that active invocations can be moved along with objects. A template for an activation record describes three things: the parameters to the operation, the local variables used by the operation, and the contents of the CPU registers.

To simplify activation record templates the Emerald compiler does not permit registers to change their template-type during an operation. A register that contains a pointer must contain a pointer for the lifetime of the invocation; however, the pointer register can point to different objects during its lifetime. This restriction is similar to the segregation of address and data registers in some architectures, but is more dynamic since the division is made for each specific operation. Without this restriction, we would need to have different templates at different points in an operation's execution – a design considered early in the project, but later abandoned as unnecessary.

## 3.4 Moving Objects

Using the addressing and implementation structure described above, the actual moving of an object is rather straightforward. Although some systems pre-copy objects to be moved for performance reasons [23], we do not believe this is necessary in the Emerald environment for several reasons. First, unlike process mobility systems, we do not copy entire address spaces. Second, many objects contain only a small amount of data. Third, even when an object with an active process is moved, we may not need to copy any code.

### 3.4.1 Moving Data Objects

Objects without active invocations are the simplest ones to move. For these, the Emerald kernel builds a message to be transmitted to the destination node. At the head of this message is the data area of the object to be moved. As we previously described, this data area is likely to contain pointers to both global and local objects. Following the data area is translation information to aid the destination kernel in mapping location-dependent addresses. For global object pointers, the kernel sends the OID, the forwarding address, and the address of the object's descriptor on the source node. For local objects, the data area is sent along with its address.

On receipt of this information, the destination kernel allocates space for the moved objects, copies the data areas into the newly allocated space, and builds a translation table that maps the original addresses into addresses in the newly allocated space. OIDs are used to locate object descriptors for existing global objects, or new object descriptors are created where necessary. The kernel then locates the template for each moved object, traverses its data area, and replaces any pointers with their corresponding addresses found in the translation table.

### 3.4.2 Moving Process Activation Records

As previously described in Section 2.5, when an object is moved the activation records for processes executing its operations must also move. This presents a particularly difficult problem: given an object to move, how do we know which activation records need to move with it? Finding the correct activation records requires a list of all active invocations for a particular object.

Several solutions are possible, but all have potentially serious performance implications. The simplest solution is to link each activation record to the object on each invocation and unlink it on invocation exit. Unfortunately, this would increase our invocation overhead by 50 percent in the current implementation. On the other hand, finding the invocations to move would require only a simple list traversal.

A second solution is to create the list only at move time. This would eliminate the invocation-time cost but would require a search of all activation records on the node. While we believe that mobility should not increase the cost of invocation, exhaustive search seems to be an unacceptable price to pay on every move.

We have therefore adopted an intermediate solution. We do maintain a list of all activation records executing in each object, as in the first solution above. However, on invocation the activation record is not actually linked into this structure. Instead, space is left for the links and the activation record is marked as "not linked", which is an inexpensive operation. When an Emerald process is preempted, its activation stack is searched for "not linked" activation records, and these are then linked to the object descriptors of their respective objects.

The search stops as soon as an activation record is found that has been linked previously. In this way, the work is only done at preemption time and its cost is related to *the difference* in stack depth between the start and end of the execution interval, not to the number of invocations performed.

An operation must still unlink its activation record when it terminates. Each return must check for a queued activation record and dequeue it before freeing the record. However, most returns will find a "not linked" activation record, in which case no work need be done.

Therefore, when an object moves we can find all activation records that must move with it merely by traversing the linked list associated with the object. These activation records are moved in a manner similar to moving data areas as described above.

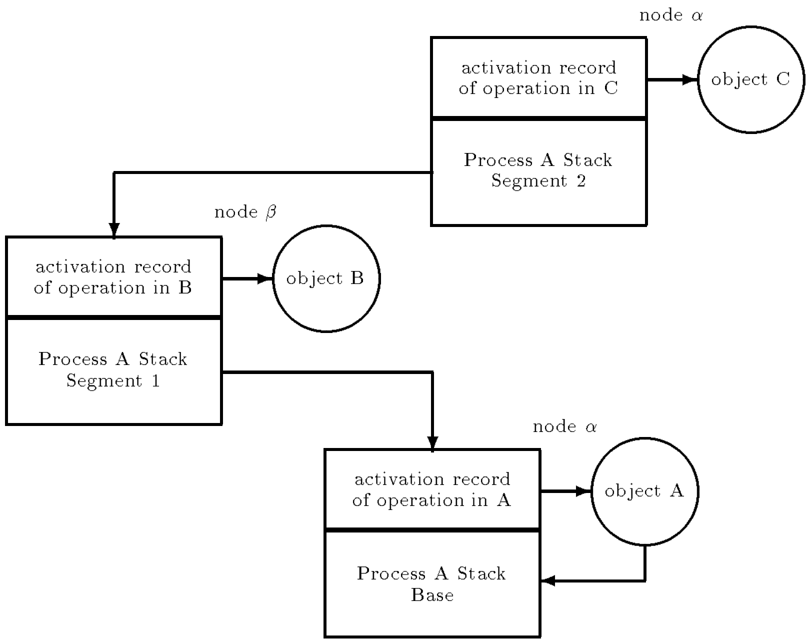


Figure 6: Process Stack after Object Move

If necessary, the activation records are removed from the stack containing them. This is accomplished by splitting the stack into (at most) three parts: the "bottom" part which remains on the source node, the "middle" part which is moved to the destination node, and the "top" part which is copied onto a new stack segment on the source node. The stack break points are found using the templates for the activation records. At each of the two stack breaks, invocation frames are modified to appear as if remote invocations had been performed instead of local invocations. Figure 6 shows the structure that would exist if object $B$ from Figure 2 were moved from node $\alpha$ to node $\beta$.

### 3.4.3 Handling Processor Registers

An additional complexity in moving Emerald processes and activation records is the management of processor registers. The Emerald compiler attempts to optimize the addressing of objects by storing local variables in registers instead of in the activation record. In this way, some of the processor registers may contain machine-dependent pointers and these must be translated when the activation record moves.

Unfortunately, the registers for a given activation record are not kept in one place. Each invocation saves in its activation record a copy of registers that will be modified by that invocation. Referring back to Figure 2, suppose that the first invocation (of $A$) and the third invocation (of $C$) both use register 5. In this case, a copy of $A$'s register 5 is saved in $C$'s activation record, as it would be in any conventional stack-based language implementation.

If object $A$ moves, its activation record will move with it. The stack will be segmented, and the rest of the stack will be left behind. Furthermore, the copy of $A$'s register stored in $C$'s activation record will be incorrect when $C$ returns, because the data that it refers to will be at

| Operation Type | Time/ms |
|---|---|
| local invocation | 0.019 |
| kernel CPU time, remote invocation | 3.4 |
| elapsed time, remote invocation | 27.9 |
| remote invocation, local reference parameter | 31.0 |
| remote invocation, call-by-move parameter | 33.0 |
| remote invocation, call-by-visit parameter | 37.4 |
| remote invocation, remote reference parameter | 61.8 |

Table 2: Remote Operation Timing

a different location on a different node.

To handle this situation, the kernel sends a copy of the registers used in an invocation along with the moving activation record. First, the kernel finds the template for the activation record in the concrete type object of the invoked object. Second, it determines which registers are used as pointers in an activation record by looking at its template. The templates for activation records have special entries for registers and for the area of the activation record where registers are saved. Third, the kernel scans the invocation stack, looking for the next activation record that has saved each of the registers. In this way, copies of the current values of the registers can be sent along with the record. On the destination node, the registers are modified using the translation table (as described in Section 3.4.1) and stored with the newly created stack segment.

For each stack segment of an Emerald process, there is a separate image of the registers. When an invocation return crosses a stack segment boundary, the registers used are those stored with the stack segment receiving control. These are the possibly translated values of registers that were computed when the stack was segmented.

# 4 Performance

We measured the performance of Emerald's mobility primitives on four MicroVAX II workstations connected by a 10 megabit/second Ethernet. These primitives have been operational for only a short time and no effort has yet been made to optimize their implementation.

Table 2 shows the elapsed time cost of various Emerald operations. The measured performance figures are averages of repeated measurements. For the simplest remote invocation, the time spent in the Emerald kernel is 3.4 milliseconds. For historical reasons, we currently use a set of network communications routines that provide reliable, flow-controlled message passing on top of UDP datagrams. These routines are slow: the time to transmit 128 bytes of data and receive a reply is about 24.5 milliseconds. Hence, the total elapsed time to send the invocation message and receive the reply is 27.9 milliseconds.

Table 3 shows the benefit of call-by-move for a simple argument object. The table compares the incremental cost of call-by-move and call-by-visit with the incremental cost of call by remote reference. The additional cost of call-by-move was 2 milliseconds while call-by-visit cost 6.4 milliseconds. These are computed by subtracting the time for a remote invocation with an

| Parameter Passing Mode | Time/ms |
|---|---|
| call-by-move | 2.0 |
| call-by-visit | 6.4 |
| call-by-remote-reference | 30.8 |

Table 3: Incremental Cost of Remote Invocation Parameters

argument reference that is local to the destination. The call-by-visit time includes sending the invocation message and the argument object, performing the remote invocation (which then invokes its argument), and returning the argument object with the reply. Had the argument been a reference to a remote object (i.e., had the object not been moved), the incremental cost would have been 30.8 milliseconds. These measurements are somewhat of a lower bound because the cost of moving an object depends on the complexity of the object and the types of objects it names.

Compared with the cost of a remote invocation, call-by-move and call-by-visit are worthwhile for even a single invocation of the argument object. As previously stated, the advantage of call-by-move depends on the size of the argument object, the number of invocations of the argument object, and the local and remote invocation costs. Emerald's fast local invocation time, about 20 *micro*seconds, easily recaptures the time for the move. Even with the current unoptimized implementation, call-by-move and call-by-visit would be worthwhile for a remote invocation cost of under 10 milliseconds.

Moving a simple data object, such as the object in Figure 4, takes about 12 milliseconds. This time is less than the round-trip message time because the reply messages are "piggybacked" on other messages (i.e., each move does not require a unique reply). Moving an object with a process is more complex; as previously stated, while Emerald does not need to move an entire address space, it must send translation data so that the object can be linked into the address space on the destination node. The time to move a small process object with 6 variables is 40 milliseconds. In this case, the Emerald kernel constructs a message consisting of about 600 bytes of information, including object references, immediate data for replicated objects, a stack segment, and general process control information. The process control information and stack segment together consume about 180 bytes.

## 5 Summary

We have designed and implemented Emerald, an object-based language and system for distributed programming. Emerald is operational on a small network of VAX computers and has recently been ported to the SUN 3[3]. Several applications have been implemented including a mail system, a replicated directory system, and a calendar system.

The goals of Emerald included:

- support for fine-grained object mobility,

[3] SUN is a trademark of SUN Microsystems, Inc.

- efficient local execution, and
- a single object model, suitable for programming both small, local data-only objects and active, mobile, distributed objects.

This paper has described the language features and run-time mechanisms that support fine-grained mobility. While *process* mobility (i.e., the movement of complete address spaces) has been previously demonstrated in distributed systems, we believe that *object* mobility, as implemented in Emerald, has additional benefits. Because the overhead of an Emerald object is commensurate with its complexity, mobility provides a relatively efficient way to transfer fine-grained data from node to node.

The need for semantic support for mobility, distribution, and abstract types led us to design a new language, and language support is a crucial part of mobility in Emerald. While invocation is location-independent, language primitives can be used to find and manipulate the location of objects. The programmer can declare "attached" variables; the objects named by attached variables move along with the objects to which they are attached. More important, on remote invocations a parameter passing mode called call-by-move permits an invocation's argument object to be moved along with the invocation request. Our measurements demonstrate the potential of this facility to improve remote invocation performance while retaining the advantages of call-by-reference semantics.

Implementing fine-grained mobility while minimizing its impact on local performance presents significant problems. In Emerald, all objects on a node share a single address space and objects are addressed directly. Invocations are implemented through procedure call or in-line code where possible. The result is that pointers must be translated when an object is moved. Addresses can appear in an object's representation, in activation records, and in registers. The Emerald run-time system relies on compiler-produced templates to describe the format of these structures. A combination of compiled invocation code and run-time support is responsible for maintaining data structures linking activation records to the objects they invoke. A lazy evaluation of this structure helps to reduce the cost of its maintenance.

Through the use of language support and a tightly-coupled compiler and kernel, we believe that our design has been successful in providing generalized mobility without much degradation of local performance.

## 6 Acknowledgements

We would like to thank Edward Lazowska and Richard Pattis for extensive reviews of early versions of this paper. We also thank Brian Bershad, Carl Binding, Kevin Jeffay, Rajendra Raj, and the referees for their helpful comments.

## References

[1] James E. Allchin and Martin S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[2] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[3] John K. Bennett. Distributed Smalltalk. In *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languagues, and Applications*, October 1987.

[4] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, October 1986.

[5] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[6] Andrew P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 181–93. ACM, December 1985.

[7] David A. Butterfield and Gerald J. Popek. Network tasking in the Locus distributed Unix system. In *USENIX Summer 1984 Conference Proceedings*, pages 62–71, 1984.

[8] Fred Douglis. Process migration in the sprite operating system. Technical Report UCB/CSD 87/343, Computer Science Division, University of California, Berkeley, February 1987.

[9] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, December 1985. Department of Computer Science technical report 85-12-1.

[10] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[11] Irene Greif, Robert Seliger, and William Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the Thirteenth Symposium on Principles of Distributed Computing*, January 1986.

[12] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[13] Carl Hewitt. The Apiary network architecture for knowledgeable systems. In *Conference Record of the 1980 Lisp Conference*, pages 107–118, Palo Alto, California, August 1980. Stanford University.

[14] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, University of Washington, January 1987. Department of Computer Science technical report 87-01-01.

[15] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 148–159, December 1981.

[16] Barbara Liskov. Overview of the argus language and system. Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, February 1984.

[17] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, October 1979.

[18] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119. ACM/SIGOPS, October 1983.

[19] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64–75, December 1981.

[20] Karen R. Sollins. Copying complex structures in a distributed system. Master's thesis, MIT, May 1979. MIT/LCS/TR-219.

[21] Eugene H. Spafford. *Kernel Structures for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, May 1986. Also Georgia Institute of Technology Technical Report GIT-ICS-86/16.

[22] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *CACM*, 25(4):246–260, April 1982.

[23] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12. ACM/SIGOPS, December 1985.

[24] Stephen Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, University of Washington, January 1987. Department of Computer Science technical report 87-01-03.

[25] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill Book Company, 1981.