The Apertos Reflective Operating System: The Concept and Its Implementation

Yasuhiko Yokote

SCSL-TR-92-014

June 26, 1992

Sony Computer Science Laboratory Inc.
3-14-13 Higashi-gotanda, Shinagawa-ku,
Tokyo, 141 JAPAN

also appeared in the Proceedings of the 1992 International Conference on Object-Oriented Programming, Systems, Languages, and Applications.

The Apertos[†] Reflective Operating System: The Concept and Its Implementation

Yasuhiko Yokote
Sony Computer Science Laboratory Inc.
Takanawa Muse Building,
3-14-13 Higashi-gotanda, Shinagawa-ku,
Tokyo, 141 JAPAN

This paper proposes a framework for constructing an operating system in an open and mobile computing environment. The framework provides object/metaobject separation and metahierarchy. In the framework, we view object migration as a basic mechanism to accommodate object heterogeneity. The relevance of the proposed framework to existing system structures is discussed. We then present a practical implementation of the Apertos operating system in this framework, where reflectors are introduced for metaobject programming and MetaCore for providing common primitives. We present some evaluation results of the Apertos operating system. We also present related work in terms of reflection mechanisms and systems.

1 Introduction

Due to the growing scale and increasing complexity of systems, we must develop a new framework for their construction. Scale and complexity motivate us to investigate open-endedness and dependability of objects. These are independent and both should be provided by a system to an appropriate degree. Advancing hardware and communication technology yields a new paradigm called *mobile computing* and enables us to witness the reality of a mobile computing environment. Here an object should be free to move around a distributed environment, because portable/mobile computers are moved frequently from one place to another. Therefore, it should also be open to evolve and adapt itself to its execution environment.

Although an object is open-ended, it is constrained by its execution environment or the real world. Therefore, its execution environment should be dependable. For example, some objects should be trusted to give objects stable services. Some objects have deadlines by which their tasks should be finished. The open-endedness and dependability properties of objects have to be individually accommodated for each object, so that object heterogeneity is accomplished.

To accommodate objects in the mobile computing environment, we have to investigate object migration. Existing systems assume stability of computers as well as objects. However, we have to consider object migration as a basis of system design. The computational field model [Tokoro 90] is a model which deals with object mobility.

Recently, it has become evident that the object-oriented technology encourages modularization, increases reusability and maintainability, gives users/programmers a single unified perspective of the system, as well as providing other advantages. This helps us to construct a system that is large in

[†] Formerly called Muse.

scale and complicated. An operating system is an example of such a system.

In our framework, an object is the only constituent of the system. We, however, cannot give an object a single set of semantics and properties. The semantics and property of an object change while an object is running and when it evolves or moves. Thus, we introduce object/metaobject separation in the operating system design. Here, an object is a container of information, whereas a metaobject defines the semantics of its behavior. In our system, each object has its own group of metaobjects, which gives an object abstract instructions or metaoperations that define the object's semantics. In this sense, a group of metaobjects can be viewed as a virtual machine, which can be optimized for an object. Since a metaobject is also an object, there are metaobjects for that metaobject. Thus, we introduce metahierarchy, within which an object and its group of metaobjects are defined.

Also, we introduce object migration as a basic mechanism of the operating system for the mobile computing environment in order to accommodate object heterogeneity. Here, object migration is defined in such a way that an object changes its group of metaobjects. That is, when an object changes its behavior or property, it migrates to another group of metaobjects. For example, for an object to acquire persistence, it migrates to a group of metaobjects that supports persistent objects. Since portable/mobile computers are frequently moving around networks, an object communicating with these computers has to change its communication protocol from local to interconnected. This occurs when the object migrates to a group of metaobjects which has suitable protocol modules to communicate with these computers.

The Apertos operating system is implemented based on the proposed framework. In the operating system, an object is defined independently from its execution environment to facilitate object migration. Here, we introduce reflectors and MetaCore to implement the Apertos operating system. A reflector is a metaobject which represents metacomputing defined by a group of metaobjects. Every reflector is represented within the reflector class hierarchy. MetaCore is a terminal metaobject in the Apertos operating system located in each computer, it has no metaspace, and it provides the common primitives for object execution.

This paper consists of the following sections. In Section 2, we present the framework characterized by object/metaobject separation, metahierarchy, and object migration. The discussion in this section includes the issues in structuring object-oriented systems and a way to accommodate object heterogeneity. We also discuss the relevance of the proposed framework to existing system structuring. Section 3 presents the implementation of the Apertos operating system based on the framework. After giving an overview of the Apertos operating system, we introduce reflectors and MetaCore for the implementation. In Section 4, we present some results of our evaluation. We present the cost of the MetaCore primitives described in Section 3 and the cost of some reflector metaoperations. We also consider a more efficient implementation of the Apertos operating system. Section 5 presents related work in terms of reflection systems and reflection in operating systems. In Section 6, we describe the current status of the implementation. Finally, Section 7 concludes this paper.

2 The Framework for Constructing an Object-Oriented Operating System

This section first presents object/metaobject separation and metahierarchy. Then, the discussion moves to the structure of the operating system in terms of object-oriented system construction and object migration. Here we also discuss a way to accommodate object heterogeneity. In summary of this section, we discuss the relevance of the proposed framework to existing operating system structures.

2.1 The Model

We believe that the object level and metalevel of abstraction should be separately described and represented within the same framework. This allows programmers to focus their attention on object programming, or describing the methods for solving problems provided by an object. Other facilities such as finding target objects and local storage management are implemented using the same framework of object programming at its metalevel. For example, an object should be described regardless of its lifetime (i.e., either temporary or permanent). The lifetime of an object is given by storage management described in the object's metalevel. For example, we can write the following simple pseudo-code to describe the object's property at its metalevel.

```
metaobject Descriptor {
  myname is    Identifier;
  execMode is  State of execution;
  instance is  Permanent segment;
  text is      Shared segment;
  queue is     Queue of messages;
  userStack is Execution stack;
  comm is      Method for communication;
  memory is    Method for storage
                management;
}
```

Here, `Descriptor` describes the object property where eight slots are defined. Each slot denotes a metaobject.

We introduce the notion of concurrent object-oriented computing [Yonezawa and Tokoro 87] to model an object. Each object encapsulates the state, methods which access the state, and a virtual processor which executes its methods. Here we introduce a *metaobject*. A metaobject is an object which defines (a part of) the behavior of that object. A virtual processor of an object can be viewed as a metaobject. A metaobject supports, for example, a way to communicate with other objects, virtual memory management and its policy, and a way to handle a faulty operation.

In the model, an object is supported by a group of metaobjects. Here we call a group of metaobjects a *metaspace*. An object has the semantics of execution that are cooperatively provided by metaobjects in a metaspace. Figure 1 shows a possible configuration of a system represented by the model[1], where a white circle represents an object and a gray area represents a metaspace. Since each

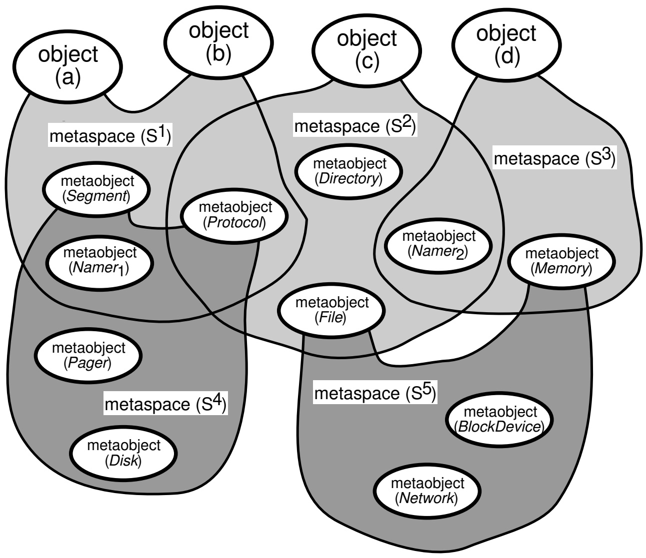

Figure 1: Object/Metaobject Separation and Metahierarchy

metaobject composing a metaspace is an object, we must introduce a metaspace for a metaobject, so that the object and its metaspace are represented within their *metahierarchy*. The relationship between an object and metaobjects composing its metaspace is relative. In the figure, metaspaces (S^1), (S^2), and (S^3) are composed of metaobjects for objects (a) and (b), object (c), and object (d), respectively. Also, metaspaces (S^4) and (S^5) are composed of metaobjects for metaobjects (*Segment*) and (*Protocol*), and metaobjects (*File*) and (*Memory*), respectively. Here, metaobjects (*Segment*) and (*Protocol*) are members of metaspace (S^1) for objects (a) and (b), i.e., (*Segment*) and (*Protocol*) are metaobjects of (a) and (b), whereas they are objects whose metaspace is (S^4).

Furthermore, since a metaspace consists of metaobjects, a metaobject can be shared between

[1] Figure 1 shows the sufficient configuration for the discussion, although the actual implementation is more complicated.

metaspaces. In the figure, metaobject (*Namer*$_1$) is shared between metaspaces (S^1) and (S^4). Also metaobjects (*Protocol*) and (*Namer*$_2$) are shared between metaspaces (S^1) and (S^2), and metaspaces (S^2) and (S^3), respectively. Thus, given an object, we can construct a metahierarchy composed of the hierarchy of metaspaces whose root is that object. For example, object (c) has metaspace (S^2), whose metaspaces are (S^4) for metaobject (*Protocol*), (S^5) for metaobject (*File*), and metaspaces for metaobjects (*Directory*) and (*Namer*$_2$)[2], and meta-metaspaces for metaobjects (*Pager*), (*Network*), etc. This means that object (c) is given the file service by metaspace (S^2) which consists of metaobjects (*File*), (*Directory*), etc. Metaobject (*File*), in turn, is given the block service by metaspace (S^5) which consists of metaobjects (*Network*) and (*BlockDevice*).

2.2 The Object-Oriented Structuring

We believe an operating system should be constructed using object-oriented technology, in which everything in a system that should be shared and protected is an object. This encourages modularization, increases reusability and maintainability, gives users/programmers a single perspective of the system, etc. Thus, it makes it possible to construct a system that is large in scale and complicated.

However, when we consider everything as an object, we encounter some difficulties. For example, it is difficult to inspect the internals of an object, because an object is protected against access from other objects. Therefore every object must have provided a method for exposing its internals which would be helpful in the implementation of a debugger. Also, it has been difficult to implement an object manager such as an invocation manager and a scheduler, because they need to access metadata such as a representation of a message and an object's state information.

Separating metaobjects from an object proposed in the previous subsection enables us to overcome the above difficulties. That is, an object is defined as a collection of metaobjects at its metalevel, so that they can access the internals of that object. Also, metaobjects are responsible for delivering messages between objects and for scheduling objects, so that they can provide optimal communication protocols and scheduling policies. Metadata for object management is an object at its metalevel. In this way, we can implement several mechanisms which help the above difficulties. These include:

- inspecting the internals of an object,
- knowing the state of an object within its execution environment,
- changing a policy of object management such as object scheduling and memory management,
- handling exceptional events such as increasing system load and missing a deadline.
- delivering a message to a destination object,
- creating a new object or deleting an existing object, and so on.

By relaxing the connection between an object and its metaspace, we can accommodate object heterogeneity. As discussed in the introductory section, each object has its own properties. Some objects are temporary and have short lifetimes. Some objects such as a name server and a directory service are shared by many client objects. Other objects depend on the underlying hardware, which include device drivers. However, the system cannot impose a single property on an object. The property of an object changes as an object grows or evolves. An object having no replica will be replicated, for example, to increase availability when it receives many message from clients on several sites. One of the sources of the object heterogeneity problem is that some objects depend on its execution environment or operating system services. There are objects such as device drivers which are, of course, inherently dependent on their underlying hardware. However, others are not. In the next subsection, we introduce object migration to facilitate the accommodation of object heterogeneity.

[2] These metaspaces are not shown in the figure.

2.3 Object Migration

We introduce *object migration* as a basic mechanism of the operating system for the open and mobile computing environment in order to accommodate object heterogeneity. Object migration is defined in such a way that an object changes its metaspace, i.e., an object travels a metahierarchy. In Figure 1, for example, we say object (a) of metaspace (S^1) migrates to metaspace (S^2), that is, object (a) changes its metaspace from (S^1) to (S^2). Also we can say object (a) migrates to metaspace (S^4).

Accommodating object heterogeneity benefits from object migration. That is, when it cannot continue its execution as it evolves or changes, it can migrate to a new metaspace to continue its execution. Object migration is performed by metaobjects in the source and destination metaspaces. Since the internals of an object are represented as metaobjects, transferring an object is equivalent to transferring metaobjects to the target metaspace.

Further, we can describe operating system services within a single framework. For example, an object can migrate to a metaspace which represents a secondary storage, when an object is to be stored onto a disk. Also, an object can migrate to a metaspace which has debugging facilities, when an object is to be debugged. In this case, since a metaobject represents the internals of an object, a debugger can be implemented as a metaobject. The following is a simple pseudo-code describing the former process:

```
object method example {
  dest := mymeta.find(secondary storage);
  mymeta.migrate(dest);
}
```

In this example, an object explicitly requests its metaspace to transfer itself to the secondary storage metaspace. In turn, the migration process in metaspace can be described by the following simple pseudo-code:

```
metaobject method migrate(dest) {
  dest.checking
        compatibility(self description);
  =ok is returned {
    dest.transfer(object descriptor);
    iterate member
             in object descriptor {
      dest.transfer(member);
    }
    return(status success);
  }
  =fail is returned {
    return(status fail);
  }
}
```

Here, before migration, compatibility between the original and destination metaspaces is examined. The detailed discussion is given in Subsection 3.2. Then, since an object descriptor, a representation of an object in a metaspace, is a collection of the references to metaobjects, the original metaspace transfers each metaobject to its destination.

2.4 Summary and Comparison

The framework proposed in this section is characterized by:

- object/metaobject separation,
- metahierarchy, and
- object migration.

Object/metaobject separation is beneficial for programmers in the sense that they can be freed from writing codes which depend on the execution environment. It overcomes some difficulties in object-oriented operating system structuring discussed in Subsection 2.2. Metahierarchy can provide discipline for metaobject programming where objects and metaobjects are separated. Object migration in this framework is a way to solve the object heterogeneity problem. Operating system services can be implemented using object migration, by introducing object migration as a basic mechanism of the system.

Although it will become difficult to meet all the requirements of user/programmers as the scale of the system grows, our framework can provide multiple execution environment for them. In Figure 1, for instance, metaspace (S^1) provides segment virtual memory management for objects, and metaspace (S^2) provides a file system capability for objects. We can also implement persistent object storage as a metaspace. An object can be persistent by moving it to that metaspace.

We can provide several novel services which can be implemented within the above framework. These include the following:

- Since a metaobject is responsible for delivering a message, it can construct the optimal network protocol for delivering a message to its target.
- Since the internals of an object are described by metaobjects and the semantics of an object is given by a metaspace, we can provide the optimal services, that is, for example, we can implement memory management suitable for object granularity [Yokote *et al.* 91a].
- Since metaspace is responsible for providing computational resources for object execution, it can provide sufficient resources to an object, so that an object with a realtime constraint is less likely to miss its deadline.
- Since multiple execution environment can be provided for users and programmers, it helps us to bridge the gap between new and old services, that is, to build a new system without sacrificing compatibility.

These have been difficult to implement in existing systems, however, our framework facilitates their implementation.

Several operating system structures have been proposed for increasing modularity, availability, and safety. These include the micro-kernel approach and the virtual machine approach. In the micro-kernel approach, operating system services are implemented as a collection of processes defined outside of the kernel. Recent operating systems takes this approach because their kernels are too complicated to implement as single module. In the virtual machine approach, the underlying hardware is virtualized as a virtual machine. In the sense that operating systems are constructed on top of the virtual machine, virtual machine can be viewed as a micro-kernel. In contrast to these structures, our framework constrains its structure to be constructed as a collection of objects which is described within the single framework characterized by object/metaobject separation, metahierarchy, and object migration. This allows us to solve the difficulties discussed in Subsection 2.2 as well as to eliminate compromise yielded by the difficulties. The compromise sometimes complicates the system structure and makes it difficult to use.

There have been object-oriented distributed operating systems such as Amoeba [Tanenbaum *et al.* 90], Chorus [Rozier *et al.* 88], and Choices [Russo *et al.* 88]. These systems support objects. In case of Amoeba, an object is a passive entity, which is accessed by an active entity, a process. The Amoeba kernel provides basically three system calls which define communication between processes. In case of Chorus, an object is an active entity, in which one or more activities, or threads, can be associated. The Chorus kernel is a micro-kernel by which minimum mechanisms such as inter-thread communication, virtual memory management, and mechanisms for realtime scheduling are implemented. In Choices, their operating system is described using object-oriented programming language, C++. In an early version of Choices, benefits of object-oriented programming were restricted to operating system programming. Recently, this restriction is eliminated, i.e., we can implement both application objects and system objects within the same framework [Campbell *et al.* 91].

In these systems, their basic abstraction is defined by their kernel. That is, a kernel defines the distinct level of abstraction, i.e., it defines the interface to application objects. In case of Chorus, for instance, object-oriented interface is provided by its kernel. A kernel thus defines the boundary between the one that employs that abstraction for objects and the other that implements that abstraction. In contrast to these systems, the Apertos operating system can be differentiated by the following. Apertos encourages concurrent object-oriented programming. Even modules that constitute a kernel in the existing systems can be defined as concurrent objects in our system. This is enabled by the framework proposed, particularly thanks to object/metaobject separation. It also increases reusability and maintainability of these modules, and frees programmers from the burden of kernel programming such as synchronization. Further, in comparison to these systems providing single interface to objects, our framework can provide multiple optimal interface to individual objects.

3 The Apertos Implementation

In this section, we present the practical implementation of the Apertos operating system based on the proposed framework. First, we present an overview of the Apertos operating system. Here we discuss some of the design goals and introduce reflectors and MetaCore for the implementation. After that, we present the implementation of reflectors which are metaobjects composing the reflector class hierarchy. Then, we present the implementation of MetaCore that provides the primitives for object execution.

3.1 Overview of the Apertos Operating System

The Apertos operating systems is targeted at the operating system for constructing very large scale, open, distributed system featured by mobile computing. It is designed based on the framework proposed in previous section. An object is the only constituent of the system, whereas each object has a metaspace consisting of metaobjects. Since the system is based on object/metaobject separation and metahierarchy, every system service is implemented by metaobjects, each of which is a member of a metaspace defined within metahierarchy. For example, an object's virtual storage is a collection of segment metaobjects and managed by a virtual segment manager metaobject. These metaobjects are also implemented as objects, i.e., there are metaspaces for these metaobjects. Here we have a question of who provides local storage for the segment metaobjects. The answer in the implementation is that we provide a paged virtual memory manager metaobject for these metaobjects. In turn, local storage for a page virtual memory manager metaobject is given by the physical memory manager metaobject.

For creation of an object, Apertos has several class systems each of which is tailored for individual programming languages. A class system is a metaspace in which metaobjects provide services such as creating an object and giving immutable properties of a class. In the implementation, a class is defined as a static and immutable template for objects [Yokote *et al.* 89]. Although a class system supports a multiple inheritance mechanism, the mechanism is only available at compile time, i.e. class hierarchy is collapsed when an object is compiled. When an object is created by operation New provided by each metaspace, the request is forwarded to a class system, and a new object is created by that class system. Then, a class system moves a new object to the original metaspace the request is sent.

We have made the following design decisions upon the Apertos design:

- We introduce a mechanism to define a metaspace which is represented within metahierarchy.
- We introduce a compile time facility to help metaobject programming.
- We introduce a mechanism to check compatibility between metaspaces.
- We define the common primitives for objects to accommodate object heterogeneity and allow us to implement object migration.

For the first item, we introduce a reflector which is a metaobject representing a metaspace. For the second item, we introduce the reflector class hierarchy within which each reflector is represented. For the third item, we introduce a `canSpeak()` mechanism which is defined in the top of the reflector class hierarchy. For the last item, we introduce MetaCore which is a metaobject having no metaspace and shared among metaspaces on a machine.

3.2 Reflectors and their Class Hierarchy

A reflector provides an object with metaoperations that are provided by metaobjects constituting a metaspace and defines a group of metaobjects. A reflector is defined within a class hierarchy, which we call the reflector class hierarchy. It is beneficial to reflector programming, i.e., we can reuse existing reflectors. The hierarchy also helps us examine reflectors' compatibility. The top of the hierarchy is mCommon[3] which is an abstract class that provides succeeding reflectors with common facilities such as object migration, a `canSpeak()` method, and descriptors that designate metaobjects representing objects.

There are several reflectors defined as subclasses of mCommon as shown in Figure 2. For example,

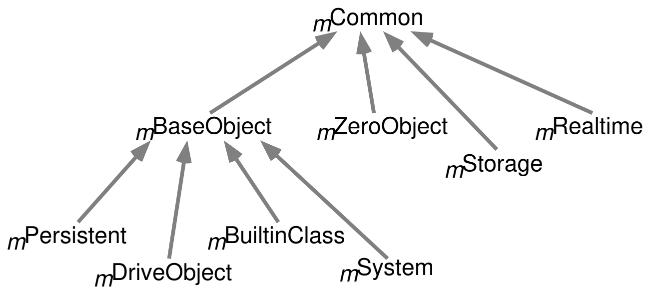

Figure 2: A Portion of the Reflector Class Hierarchy

mDriveObject is a reflector which provides metaoperations for interrupt handler programming of device drivers. In the implementation, it is implemented so that the duration of interrupt handling is as brief as possible. mBuiltinClass is a reflector which represents a metaspace for class objects, so that it provides metaoperations to manage classes such as relocating code segments and inspecting the internals of a class. mRealtime is a reflector which provides objects with a realtime scheduling facility. We can define a new reflector which has new metaoperations as a subclass of existing ones.

Figure 3 shows a simple view of the implementation of the Apertos operating system using reflectors[4]. mZeroObject is a metaspace for reflectors mSystem, mBaseObject, and mRealtime. mRealtime is a metaspace for a realtime object whereas a realtime clock metaobject is a member of metaspace mRealtime, whose metaspace is mSystem. In the succeeding subsections, reflectors mZeroObject and mBaseObject are presented in detail.

For checking compatibility of a metaspace upon object migration, a `canSpeak()` mechanism is introduced. Here, `canSpeak()` is a common method for all reflectors which examines compatibility of reflectors or metaspaces upon object migration. When an object migrates to a metaspace, a metaspace should be compatible with the original metaspace. Compatibility is defined in such a way that when an object can continue execution after it migrates to a metaspace, the original and destination metaspaces are totally compatible; when an object can continue its execution with some restrictions after it migrates to a metaspace, the original and destination metaspaces are partially compatible; when an object cannot continue execution after it migrates to a metaspace, the original and a destination are incompatible.

In the implementation, compatibility depends on the reflector class hierarchy and is determined by a `canSpeak()` method. Several system services can be discussed from the viewpoint of the reflector compatibility. For example, disconnected operations for portable computers, which are operations performed while portable computers are disconnected from a network[5], can be described by object migration and reflector compatibility. That is, before starting disconnected operations, objects migrate to a portable computer to continue execution. Here, a metaspace on a portable computer is partially compatible with the original metaspace, so that the objects incur a limitation of services. Also, since the metaspace on the portable computer

[3] In this paper, we use notation mReflector to name a reflector.

[4] Since it is difficult to depict the actual implementation due to its complication, the figure shows so simple as to understand the implementation.

[5] For example, Coda [Kistler and Satyanarayanan 91] supports disconnected operations for its file system.

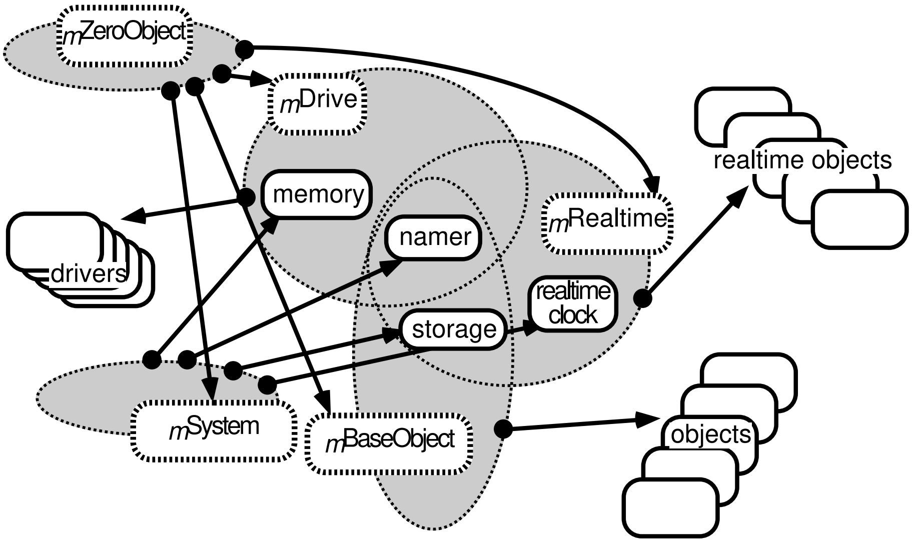

Figure 3: The Simple View of the Apertos Implementation using Reflectors

knows it is only partially compatible for certain objects, it can take appropriate action when objects cannot continue their execution.

3.2.1 mZeroObject

The mZeroObject reflector provides facilities for *reflector programming*, having the operations shown in Table 1. These operations are used by reflectors only. In the current implementation of mZeroObject, it provides both synchronous and asynchronous communication facilities using continuations (Call, Force, Send, Reply, and Return). A continuation designates a point where execution is resumed with the replied result. A continuation enables a reflector to accept another request after issuing commands for blocking operations such as Call and Force. This also reduces the so called remote delay [Liskov *et al.* 85].

mZeroObject has no New and Delete operations, therefore a new reflector must be created by the following procedures. First, a new reflector is created by a metaspace which has the ability to create a new object. Then, it must migrate to mZeroObject using the Migrate operation. The reason why we adopt such an implementation is that mZeroObject is intended to be free from the details of memory management, and therefore, it has no ability to manage virtual storage.

The protocol of migration can be divided into two steps. The first step is to receive a descriptor from a reflector that creates a new reflector, which includes name of metaobjects composing a migrating reflector and sufficient information to resume object execution. At this step, `canSpeak()` is invoked to examine whether it is a reflector. The second step is to transfer the contents of a reflector. In the implementation, all of reflectors on the same machine occupy the same address space, so that the actual transfer is not made, i.e., mZeroObject installs a receiving descriptor as a new reflector.

mZeroObject maintains the internal scheduler to control execution of reflectors. It also maintains a map of an object and its reflector (or metaspace), so that a reflector can send a message to a target object without knowing its reflector. Further, each host has only one mZeroObject in the current implementation. mZeroObject is intended to be a metaspace for reflectors in this way.

3.2.2 mBaseObject

The mBaseObject reflector provides facilities for *concurrent object-oriented programming.* It has the

Table 1: The mZeroObject Interface

primitive	*description*
Call	invokes a method defined in the target object (a metaobject or reflector). This activates the internal scheduler that determines the reflector to be next activated.
Force	performs the same operation as Call. However, it never activates the internal scheduler. That is, a method of the target is directly invoked if the target is not busy. Otherwise, an error is returned.
Send	performs the same operation as Call except that the object that issues operation Send continues execution.
Reply	delivers the result back to the point that the continuation designates. It activates the internal scheduler to find the reflector to be next activated.
Return	performs the same operation as Reply except that it never activates the internal scheduler. If the reflector that is to be activated by Return is busy, an error is reported.
Preempt	takes CPU execution priority away from the running reflector. It is usually requested by a reflector of an interrupt handling object.
Exit	terminates reflector execution. This causes the internal scheduler to pick up another reflector to be scheduled.
Find	returns a map between an object and its reflector.
Install	sets up a map between an object and its reflector.
Migrate	moves a reflector created by another reflector (or metaspace) to this reflector, mZeroObject.

operations shown in Table 2. In the current implementation of mBaseObject, it provides both synchronous and asynchronous communication facilities with remote procedure call semantics. In contrast to mZeroObject, it provides the New and Delete operations. Since mBaseObject encourages concurrent object-oriented programming, an object cannot make a request for raw memory allocation such as `malloc()`, i.e., New and Grow are the only ways to allocate memory.

Reflector mBaseObject has two interface: one is the above interface which is used for objects, and the other is the following interface shown in Table 3 which is used for other reflectors and metaobjects sending a message to this reflector. In addition, mBaseObject makes public several continuations for the requests sent by New, Delete, Grow, and Shrink in Table 2.

The migration procedure between reflectors is the following. It is similar to the one of mZeroObject. That is, the protocol of migration can be divided into two steps. The first step is to transfer a descriptor to the target reflector, which includes name of metaobjects composing a migrating object and sufficient information to resume object execution. At this step, `canSpeak()` is invoked to examine compatibility. The second step is to transfer the contents of an object, i.e. metaobjects, either eagerly or lazily. This step is the responsibility of memory management metaobjects. If two metaspaces are on the same host, they are usually not transferred.

3.3 MetaCore

MetaCore is a terminal metaobject which has no metaspace and is similar to a micro-kernel in existing systems. It provides all type of objects with the common primitives facilitating object/metaobject separation and object migration. We have laid down the following design goals:

- MetaCore should have no knowledge of object identity, because object identity should be

Table 2: The mBaseObject Interface to Objects

primitive	*description*
Call	invokes a method defined in the target object. This activates the internal scheduler that determines an object to be activated next. If the target is not ready to accept the request, it is stored into the queue maintained in a descriptor that is managed by mBaseObject.
Send	performs the same operation as Call except that the object that initiates the Send operation continues execution.
Reply	delivers the result back to the sender object. It activates the internal scheduler to find an object to be activated next. If a request is pending for the object initiating the Reply operation, it is scheduled to be processed.
New	creates a new object.
Delete	removes an existing object.
Grow	makes an object bigger.
Shrink	makes an object smaller.
Yield	gives up the possession of the underlying CPU voluntarily. The internal scheduler selects an object to resume execution.
Find	returns a map between an object and its reflector.
Install	sets up a map between an object and its reflector.
Migrate	moves an object to another metaspace.

given by a metaobject.

- The execution time for all of the primitives provided by MetaCore should be predictable. For example, code for searching a data structure should not be included.
- The underlying CPU should be virtualized to help object migration.
- MetaCore should be implemented as small as possible for portability and maintainability.

For the first item, object identity is given by metaobjects, called *namers*, which compose the hierarchical structure. We have introduced a hierarchical naming scheme for objects which can distinguish two or more objects without unique identifiers [Fujinami and Yokote 92]. For the last item, in the current MC68030 implementation, MetaCore occupies 7456 bytes for text, 1220 bytes for initialized data, and 4268 bytes for uninitialized data.

To meet these goals, we have introduced the *Context* structure that represents the underlying CPU. Figure 4 shows the MetaCore implementation. MetaCore maintains the *Context* structure associated with an object by a reflector, that is, an object requires the *Context* structure to execute its method. One object is associated with one *Context* in the current implementation (a gray arrow specifies this association in Figure 4). Although a reflector holds the "meta-of" link, *Context* also holds the same link as a cache (in Figure 4, a dashed arrow represents the "meta-of" link). The internals of the *Context* structure can only be accessed by MetaCore. An object can examine the internals of *Context* using the primitives provided by MetaCore. The *Context* structure currently maintains the information shown in Table 4. Although MetaCore is not concerned with *Context* scheduling, it has to maintain its state to avoid multiple execution of the same *Context*. In the implementation, MetaCore maintains four states, "free" that has no association with an object, "busy" that is running on the underlying CPU, "suspend" that causes *Context* execution to wait to be resumed when receiving a hardware interrupt, and "dormant" that is ready to accept a new request.

Table 3: The mBaseObject Interface to Metaobjects

primitive	*description*
Deliver	delivers an incoming message sent by other reflectors. This is provided for inter-reflector (or inter-metaspace) communication.
Migrate	handles an incoming request of object migration.
Preempt	takes away CPU execution priority, so that the running object is interrupted and the internal scheduler of mBaseObject picks up another object to be executed. This is provided to support an interrupt event, and usually used by mZeroObject.

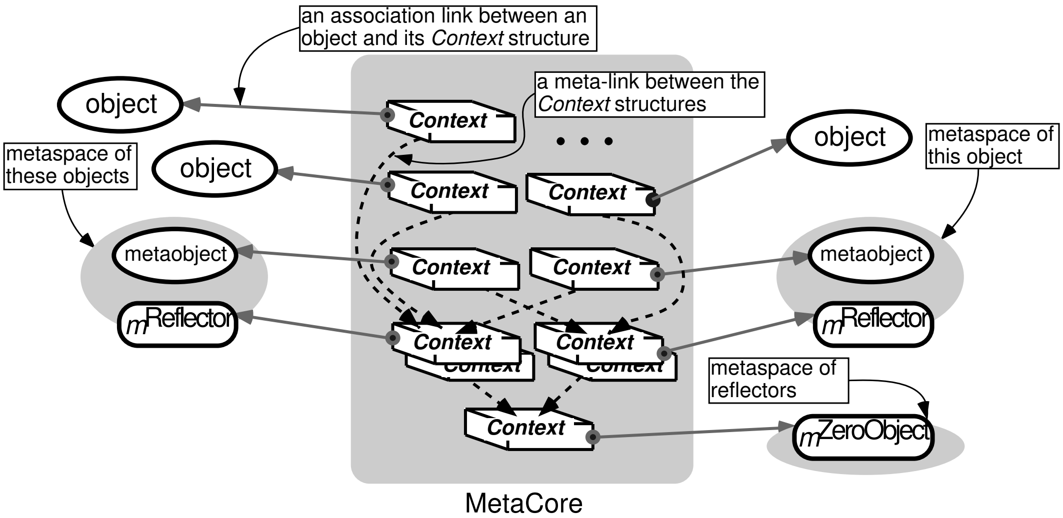

Figure 4: The MetaCore Implementation

In retrospect, we can review similar mechanisms in [Levin *et al.* 75] and recently in [Anderson *et al.* 91] and [Marsh *et al.* 91] in the sense that a kernel does not handle scheduling policies. MetaCore provides the public primitives shown in Table 5. We can use these primitives using a C++ interface as shown in Figure 5. The interface is so designed as to be generic for any CPU architecture. When a programmer wants to make a request for metaspace, for example, by primitive M, to MetaCore, he/she simply writes:

```
MessageM msg (MZERO_MIGRATE, pMigMsg);
if (msg.Act ().state == mcSUCCESS)
  // Primitive M is successfully done.
else
  // Primitive M has failed.
```

The above code is usually hidden from programmers by libraries or compilers. Here, structure `MessageM` is filled with the following values: member `code` has a value specifying the request is primitive M; member `selector` has a value of `MZERO_MIGRATE`, which is the name of a method defined in the reflector to which the request is sent; member `message` has an address, `pMigMsg`, which is delivered to a reflector; and member `inherit` has a value `FALSE`, which designates priority inheritance upon *Context* activation.

3.4 Summary

The Apertos operating system is totally based on the framework proposed in the previous section. The current implementation has paid as much attention to keep obeying the framework as possible and has made a sacrifice of its performance. In Subsection 4.3, we discuss several ways to improve

Table 4: The Elements of *Context* Structure

elements	*description*
name	Name of structure *Context* used to designate *Context* by a reflector.
link	Two links are supported: one is a link to the meta-*Context* structure representing metacomputing, and the other is a link to *Context* representing object computing that has recently activated metacomputing.
priority	Priority of CPU execution which is installed in the interrupt priority mask field (in the case of MC68030) before *Context* activation.
working area	This is provided for user and system modes of *Context* execution.
register set	This is used to save/restore hardware registers when *Context* changes.
mode	Mode of *Context* execution. In the case of MC68030, two modes, user and system, are supported.
state	State of *Context* execution.
entry table	This maintains addresses to start *Context* execution.

overall system performance.

The features of the Apertos implementation is summarized by:

- reflectors and their reflector class hierarchy, which provide users/programmers with protocols for metaobject programming.
- MetaCore, which defines the common primitives for object execution.

The several advantages of the system stem from the Apertos implementation. The primitives provided by MetaCore is generic but sufficient for constructing the operating system for mobile computing environment. This also allows us the quick implementation for a new architecture.

4 Evaluation

We have measured the cost of the primitives provided by MetaCore and the cost of metaoperations particularly provided by reflectors mZeroObject and mBaseObject. The results from the measurement of primitive M and operation Call of mBaseObject are ramified into several steps. We also offer some hints for the efficient implementation of the Apertos operating system. These results have been measured on a Sony PWS-1550 workstation which has a 25MHz MC68030 CPU with a minimum 4MB of physical memory.

4.1 The Cost of MetaCore Primitives

Table 6 shows the results of our measurement of the primitives of MetaCore. In the table, primitive R has two figures: (D) presents the cost of R when the *Context* state is "dormant" and (S) presents the cost when the *Context* state is "suspend." When the *Context* state is "suspend," the *Context* values do not need to be restored, so that it shows a lower cost. Also, primitives CNew, CSetAttribute, and CGetAttribute vary in cost according to their argument, and the figures shows minimum values. In contrast to these primitives, the M and R primitives are guaranteed to take the fixed time as shown in the table.

Table 6: The Cost of MetaCore Primitives

primitive	*cost (in μsec)*
M	65
R	84(D), 49(S)
CNew	71~
CDelete	94
CBind	45
CUnbind	32
CSetAttribute	51~
CGetAttribute	48~

Table 7 shows the time interval between the oc-

Table 5: The MetaCore Interface

primitive	*description*
M	makes a request for metacomputing. This causes the execution of *Context* that is designated by the "meta-of" link. M causes object execution to stop, which is then resumed by primitive R.
R	resumes object execution. In contrast to primitive M, R can resume execution of any *Contexts* that M stops.
CNew	creates a new *Context* structure. CNew can take an argument which designates the initial information of a new *Context* structure.
CDelete	designates a *Context* state as "free." This *Context* structure is reallocated by CNew.
CBind	associates *Context* with a hardware interrupt. It requires an argument which includes a message to be delivered to an interrupt handling object. When a hardware interrupt occurs, the active *Context* is suspended and its associated *Context* is immediately activated to execute a method of an interrupt handling object.
CUnbind	removes the association that is made by CBind.
CSetAttribute	installs specified parameters in *Context*.
CGetAttribute	retrieves specified parameters from *Context*.

currence of an interrupt and starting of the handler. We measured the cost of an exception named Illegal Instruction. The ExceptionHandler is always invoked by underlying hardware. It supplies a message (`HardMessage` in the implementation) and activates *Context* that handles exceptions. This process eventually activates an exception handling object. The cost of ExceptionHandler depends on the format of the exception stack frame. In the case of the above measurement, MC68030 pushed 8 bytes of data onto a system stack.

Table 7: The Cost of Exception Handler of MetaCore

primitive	*cost (in μsec)*
ExceptionHandler	76~

4.2 The Cost of Metaoperations

Table 8 shows the cost of the operations provided by $m^{\text{ZeroObject}}$. These figures include the overhead of a stub procedure. Table 9 shows the cost of the operations provided by $m^{\text{BaseObject}}$. Further,

Table 8: The Cost of $m^{\text{ZeroObject}}$ Operations

operation	*cost (in μsec)*
Call	445
Force	246
Send	495
Reply	475
Return	214
Exit	246
Find	374
Install	556
Migrate	840

Table 9: The Cost of Some $m^{\text{BaseObject}}$ Operations

operation	*cost (in μsec)*
Call/Reply roundtrip	423
Send	374

```
struct Message {
        magicword       magic;  // magic
        longword        code;   // primitive code
        // initializing
        Message (longword c);
};
struct MessageM : Message {
        longword        selector;       // selector of meta-computing
        void*           message;        // message to meta-computing
        Boolean         inherit;        // TRUE inherits the priority
        // initializing
        MessageM (longword select = UNDEF, void* pMsg = NULL,
                Boolean in = FALSE);
        // primitive interface to MetaCore
        SysError        Act ();
        SysError        M (longword select, void* pMsg = NULL,
                                Boolean in = FALSE);
};
struct MessageR : Message {
        // Context to reflect the result of meta-computing
        CName           context;
        // selector of object-computing quickly to be resumed
        longword        selector;
        // message to object-computing to be resumed
        void*           message;
        // TRUE inherits the priority
        Boolean         inherit;
        // initializing
        MessageR (CName ctxt = UNDEF, longword select = UNDEF,
                void* pMsg = NULL, Boolean in = FALSE);
        // primitive interface to MetaCore
        SysError        Act ();
        SysError        R (CName ctxt, longword select = UNDEF,
                                void* pMsg = NULL, Boolean in = FALSE);
};
```

Figure 5: A Portion of the Message Structure to MetaCore

Table 10 shows the cost of object creation. Here, since it is a function of the object size, we show two examples. In the current implementation, Apertos assigns the same address space to one or more objects, so that data in the text and data segment should be relocated to their own addresses upon object creation. These operations are not guaranteed to take the same time: they depend on the number of objects, the size of the objects, etc.

Table 10: The Cost of Object Creation (Operation New of mBaseObject)

size in byte(text+data+bss)	*cost (in msec)*
9828+2368+4188	25.4
13500+27100+4188	33.3

4.3 Toward a More Efficient Implementation

We divide a procedure of primitive M into the following steps:

1. handling a trap: a procedure to transfer control to *Context* of MetaCore using the trap instruction, which includes the overhead of a stub procedure.
2. saving registers: a procedure to store hardware registers to register set of *Context*.

3. checking a message: a procedure to check whether a message is correct.
4. finding *Context*: a procedure to find *Context* designated by `MessageR`. This procedure is only valid for primitive R.
5. checking validity of meta-*Context*: a procedure to check whether the values of *Context* are correct.
6. restoring meta-*Context*: a procedure to load appropriate values into *Context* that is to be resumed.
7. restoring registers: a procedure to load values stored in *Context* into hardware registers.
8. resuming execution: a procedure to transfer control to a metaobject, i.e. meta-*Context*, which includes the overhead of a stub procedure.

Since the *Context* structure has a "meta-of" link, it does not need to search for meta-*Context*. Also, we divide primitive R into similar steps except for the step that searches for *Context* (step 4). Table 11 shows the result of our measurement of the above steps.

The steps that take time to process are steps 1, 2, 7, and 8, which account for over 40% of the total processing time. It is difficult to eliminate these steps, particularly steps 2 and 7, even by using an assembler language, because primitive M causes *Context* to change. However, we can eliminate the trap instruction by using the similar mechanism to inline caching when it does not change the CPU execution mode.

From the table, step 5 accounts for roughly one quarter of the total processing time, so that a way to reduce overall cost is to reduce this step. In the current implementation, the content of the *Context* structure is guaranteed upon its activation, so that step 5 takes a certain amount of time. However, this step can be reduced by postponing the checking, that is, we can insert a mechanism to compensate for the incorrect *Context* using fault detection mechanisms of the underlying CPU such as page fault handling.

In comparison to the cost of UNIX[6] system calls, primitive M takes 65μsec, which roughly corresponds with a null system call in UNIX. Our measurement using NEWS-OS 4.1C (which resembles UNIX 4.3 BSD) shows that it takes 64μsec. That is, primitive M incurs the same cost as UNIX. Also, in the case of the ARTS kernel, a null system call takes 30μsec [Ishikawa 91a], which is two times faster than primitive M. This is the same duration as the sum of steps 1, 2, 7, and 8 of primitive M.

We divide the procedure of the $m^{\text{ZeroObject}}$ Call operation into the following steps:

1. handling a stub: a procedure to prepare for the invocation of the Call operation.
2. locating a target descriptor: a procedure to find the descriptor that designates a receiver reflector.
3. enqueuing a target descriptor: a procedure to add a target descriptor to the ready queue.
4. dequeuing a descriptor: a procedure to remove a descriptor whose execution is to be resumed from the ready queue.
5. preparing to resume execution: a procedure to restore appropriate values to a descriptor and construct `MessageR` in order to resume execution (with a message if any) using primitive R.

In the implementation, $m^{\text{ZeroObject}}$ maintains the ready queue that is used to schedule reflectors. We have measured the above steps and show the results in Table 12. Steps 1 and 5 account for one half of the total processing time. That is, a considerable amount of time is wasted for stub handling in the current implementation. Since there is a gap between an object of C++ and an object of the Apertos operating system, we need this small stub in the current implementation. However, it can be reduced by a stub generator or a compiler.

Further, [Masuhara *et al.* 92] proposes the several ways to implement a reflective programming language efficiently. Although their system structure is different from ours, some mechanisms such as light-weight objects and non-reifying objects can

[6] UNIX is a registered trademark of AT&T Bell Laboratories.

Table 11: Break Down of Measurements for MetaCore M and R Primitives Costs

step	*M (in μsec)*	*M, ratio (%)*	*R (in μsec)*	*R, ratio (%)*
1: handling a trap	5	7.7	9	10.7
2: saving registers	8	12.3	9	10.7
3: checking a message	3	4.6	4	4.8
4: finding *Context*	–	–	6	7.1
5: checking validity of *Context*	18	27.7	20	23.8
6: restoring *Context*	15	23.1	16	19.1
7: restoring registers	10	15.4	11	13.1
8: resuming execution	6	9.2	9	10.7
total	65	100	84	100

Table 12: Break Down of Measurements for *m*ZeroObject Call Operation Cost

step	*cost (in μsec)*	*ratio (%)*
1: handling a stub	96	21.6
2: locating a target descriptor	41	9.2
3: enqueuing a target descriptor	159	35.7
4: dequeuing a descriptor	20	4.5
5: preparing and resuming execution	129	29.0
total	445	100

be applied to a more efficient implementation of the Apertos operating system.

5 Related Work

The framework proposed in Section 2 is related to reflective computing in programming languages presented in [Smith 84], [Maes 87], etc. in the sense that the model of an object is also represented within the same framework of an object and described as metaobjects. Reflective computing in the proposed framework is defined in such a way that it is a process to improve or alter the object's behavior using the object migration mechanism. In the implementation, reflectors provide objects with metaoperations for reflective computing. Thus, reflectors and their reflector class hierarchy describe protocols for reflection programming called metaobjects protocols [Kiczales *et al.* 91]. In this section, we first discuss the relevance of the proposed framework to reflection systems. We then discuss the relevance of the framework to reflection in operating systems.

5.1 Reflection Systems

Many systems that are based on a reflective architecture have been proposed particularly in programming languages such as 3-Lisp [Smith 84], 3-KRS [Maes 87], Self [Hölzle *et al.* 90], and CLOS [Kiczales *et al.* 91]. We can compare these systems from the viewpoint of various system structures, i.e., how metaobjects are organized, how classes are organized, how an object and its metaobject correspond, how classes and metaobjects are differentiated, etc. Since these systems are constructed within a programming language framework, there is a limitation on the management of computing resources. Also, metaprogramming is achieved in a single programming language.

ABCL/R2 [Matsuoka *et al.* 91] is an example proposal which is equipped with two reflection structures, individual-based and group-wide ones. While, in the former, each object has a causal connection link to its own single metaobject, in the latter, each object has a causal connection link to a metagroup, a group of metaobjects. Here, computing resources are cooperatively managed by a group of metaobjects as in Apertos. AL-1 [Ishikawa 91b] is another proposed example. In his paper, a two-dimensional hierarchical reflection model (recently a multi-model reflection framework) was proposed. The model introduces the notion of metafloor which describes an implementation of the above floor. A resource metafloor is responsible for managing computing resources. These approaches enable us to describe the management of computing resources in a programming language framework.

Applying a reflective computing technology is not limited to programming languages. Silica [Rao 91] is an example applied to a CLOS-based window system. Apertos is the first example which applies the notion of reflective computing to a distributed object-oriented operating system. We can find many other reflection systems in [Ibrahim 91].

In contrast to the above systems, the model proposed for the Apertos operating system is the first approach to defining object behavior as a collaborative group of metaobjects. We first proposed it in [Yokote *et al.* 89] and elaborated in [Yokote *et al.* 91b]. Here, metaspace is defined as a group of metaobjects. Thus some metaobjects can be shared among metaspaces. This contributes to the implementation of a metaobject which needs a common view among several objects, for instance, a namer metaobject is shared among metaspaces, so that it can define one naming domain per host. This also enables us to implement a metaobject which can access an inherently single object such as a console device.

We are convinced that class hierarchy and metahierarchy should be separately defined. In Apertos, a class defines the structure of an object, while a metaobject defines the computation of an object. Thus, a class is introduced to accommodate the structural heterogeneity such as programming languages and underlying hardware. A metaobject is introduced to accommodate the semantic heterogeneity such as difference in communication and difference in lifetime.

5.2 Reflection in Operating Systems

An operating system is inherently reflective: it has facilities to inspect system data structure maintained by a kernel and to inspect the internals of a process, such as `/dev/kmem` in UNIX. However, these have a limitation in their use, that is, since they are mechanisms provided by an operating system kernel, it is difficult to alter their behavior.

Also, in virtual memory management, we can specify its management policy. UNIX provides `madvise()` for it. Recently, Mach [Tevanian 87] and Chorus [Abrossimov *et al.* 89] introduce user-level virtual memory management as an external pager. Using an external pager approach enables us to write a code for an application specific policy for memory management. Further, in realtime operating system, we can describe a realtime scheduling policy which determines the process/thread next to be executed. ARTS [Tokuda and Mercer 89] is an example of such a system providing policy/mechanism separation, where we can write a policy module which uses mechanisms given by its kernel to choose the thread to be run.

In contrast to these approaches, the framework proposed in this paper has significant advantages. In this framework, the system is constructed based on object/metaobject separation and metahierarchy. This increases modularity of the system. That is to say, the system components can be easily modified and reused to create a new service. From the point of reflective computing, the primitives shown in Table 5 are atomic operations to implement reflection mechanisms. Primitive M initiates metacomputing that may make an object alter its behavior. Primitive R terminates metacomputing and resumes object execution. Primitives CNew through CGetAttribute maintain the link between an object and its metaspace or reflector. This link is not fixed so that we can freely reconnect an object with its metaspace. This helps us implement object migration. Without reflection mechanism, we encounter the difficulties such that we cannot accommodate object heterogeneity, we cannot adapt an object to its execution environment given by mobile computers, and we cannot implement an optimal service for evolving objects.

6 The Current Status and Future Work

The Apertos operating system has been implemented on a Sony PWS-1550 workstation. MetaCore has been stably operational for a year. Reflectors $m^{\text{ZeroObject}}$, $m^{\text{BaseObject}}$, and $m^{\text{DriveObject}}$ are also running. Metaobjects such as managing virtual memory, storage devices, and a console device are also available. The system is implemented using the AT&T C++ programming language [Ellis and Stroustrup 90]. Libraries for programming in the Apertos system are provided.

We are also implementing our system on a MIPS R3000 based workstation. Since the MetaCore interface is the same as the MC68030 version of the system, reflectors and metaobjects for the CISC implementation are compatible with the RISC implementation.

The programming interface of the Apertos operating system is not yet satisfactory. At present, we have to write a simple stub by hand to make a request for metacomputing. MC++, an extension of C++, is intended to be designed to facilitate reflective programming in C++. Also, writing reflectors is different from writing objects in the sense that reflectors require two interface: one for objects and the other for reflectors and metaobjects. These two interfaces should be elegantly incorporated in a programming language.

Further, the `canSpeak()` method is restrictive in that it uses a reflectors class hierarchy to examine compatibility. We need to continue the investigation of this mechanism.

7 Conclusion

We proposed the framework of the object-oriented operating system for an open and mobile computing environment. The framework is characterized by object/metaobject separation, metahierarchy, and object migration. Object/metaobject separation and object migration help users and programmers to accommodate object heterogeneity. These enable us to provide the mechanisms such as changing a communication paradigm, inspecting the internals of an object for a debugger, and changing resource management policy. Metahierarchy provides discipline for programming in object/metaobject separation. Object migration is a basic mechanism of the operating system in order to accommodate object heterogeneity. These hide the underlying implementation from an object and increases mobility of object, so that these contribute to the realization of an open and mobile computing environment.

We introduced reflectors for metaobject programming and MetaCore for providing the common primitives for object execution. Reflectors are defined within their reflector class hierarchy, which describe protocols for metaobject programming. MetaCore is a terminal metaobject having no metaspace located at each computer and a very small kernel which implements the primitives of the Apertos operating system. Further, we presented the cost of the MetaCore primitives and some metaoperations, and discussed ways to further improve the efficiency of the Apertos operating system.

The prototype implementation of the Apertos operating system is available to anyone for nonprofit purposes[7].

Acknowledgments

I offer my sincere thanks to Prof. Mario Tokoro, the director of the Sony Computer Science Lab-

[7] Information about the distribution of the Apertos operating system is available through anonymous FTP from `scslwide.sony.co.jp` (133.138.199.1).

oratory Inc. He and I have been collaboratively investigating the conceptual design of the Apertos operating system. I also give my thanks to members of the Apertos project at Sony Computer Science Laboratory Inc. and to Dr. Hideyuki Tokuda of Carnegie Mellon University. Several discussions with these people were helpful to us in designing the structure of the system and its reflector classes. Further, I thank Dr. Takao Tenma and Mr. Nobuhisa Fujinami of Sony Computer Science Laboratory Inc. who read my draft paper and gave me many useful comments for improvement. I also thank Mr. Atsushi Mitsuzawa who helped me to measure the cost of the primitives of the Apertos operating system. Finally, I thank reviewers of this paper. Their valuable comments were useful for improving this paper.

References

[Abrossimov *et al.* 89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pp. 123–136, December 1989.

[Anderson *et al.* 91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 95–109, October 1991.

[Campbell *et al.* 91] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 9–15. IEEE Computer Society Press, October 1991.

[Ellis and Stroustrup 90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Fujinami and Yokote 92] Nobuhisa Fujinami and Yasuhiko Yokote. Naming and Addressing of Objects without Unique Identifiers. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992. also appeared in SCSL-TR-92-004 of Sony Computer Science Laboratory Inc.

[Hölzle *et al.* 90] Urs Hölzle, Bay-Wei Chang, Craig Chambers, and David Ungar. *The Self Manual — Version 1.0*, June 1990.

[Ibrahim 91] Mamdouh H. Ibrahim, editor. *Proceedings of the OOPSLA'91 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[Ishikawa 91a] Yutaka Ishikawa. Position Paper on I-WOOOS, 1991. appeared in the collection of position papers for the International Workshop on Object-Orientation in Operating Systems.

[Ishikawa 91b] Yutaka Ishikawa. Reflection Facilities and Realistic Programming. *ACM SIGPLAN NOTICES*, Vol. 26, No. 8, August 1991.

[Kiczales *et al.* 91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[Kistler and Satyanarayanan 91] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 213–225, October 1991.

[Levin *et al.* 75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. POLICY/MECHANISM SEPARATION IN HYDRA. In *Proceedings of the 5th ACM Symposium on Operating System Principles*, pp. 132–140. ACM Press, November 1975.

[Liskov *et al.* 85] Barbara Liskov, Maurice Herlihy, and Lucy Gilbert. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. Technical Report Programming Methodology Group Memo 41-1, Massachusetts Institute of Technology, September 1985.

[Maes 87] Pattie Maes. COMPUTATIONAL REFLECTION. Technical Report TR-87-2, VUB AI-LAB, 1987.

[Marsh *et al.* 91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 110–121, October 1991.

[Masuhara *et al.* 92] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992.* ACM Press, October 1992.

[Matsuoka *et al.* 91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Proceedings of ECOOP'91 European Conference on Object-Oriented Programming*, pp. 231–250, July 1991. Lecture Notes in Computer Science 512.

[Rao 91] Ramana Rao. Implementational Reflection in

Silica. In *Proceedings of ECOOP'91 European Conference on Object-Oriented Programming*, pp. 251–266, July 1991. Lecture Notes in Computer Science 512.

[Rozier *et al.* 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems*, Vol. 1, No. 4, pp.305–370, Fall 1988.

[Russo *et al.* 88] Vincent Russo, Gary Johnston, and Roy Campbell. Process Management and Exception Handling in Multiprocessor Operating Systems using Object-Oriented Design Techniques. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1988*. ACM Press, September 1988. also appeared in SIGPLAN NOTICES, Vol.23, No.11.

[Smith 84] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, January 1984.

[Tanenbaum *et al.* 90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, Vol. 33, No. 12, pp.46–63, December 1990.

[Tevanian 87] Avadis Tevanian, Jr. Architecture Independent Virtual Memory Management for Parallel and Distributed Environment: The Mach Approach. Technical Report CMU-CS-88-106, Department of Computer Science, Carnegie Mellon University, December 1987.

[Tokoro 90] Mario Tokoro. Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment. In *Proceedings of the 2nd IEEE Workshop on Future Trends in Distributed Computing Systems*, September 1990. also appeared as Technical Report SCSL-TR-90-006.

[Tokuda and Mercer 89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A Distributed Real-Time Kernel. *Operating Systems Review*, Vol. 23, No. 3, pp.29–53, July 1989.

[Yokote *et al.* 89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A Reflective Architecture for an Object-Oriented Distributed Operating System. In *Proceedings of ECOOP'89 European Conference on Object-Oriented Programming*, July 1989. also appeared in SCSL-TR-89-001 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 91a] Yasuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. Reflective Object Management in the Muse Operating System. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 16–23. IEEE Computer Society Press, October 1991. also appeared in SCSL-TR-91-009 of Sony Computer Science Laboratory Inc.

[Yokote *et al.* 91b] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse Object Architecture: A New Operating System Structuring Concept. *ACM Operating Systems Review*, Vol. 25, No. 2, pp.22–46, April 1991. also appeared in SCSL-TR-91-002 of Sony Computer Science Laboratory Inc.

[Yonezawa and Tokoro 87] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.