*Roy H. Campbell, Nayeem Islam, David Raila and Peter Madany*

# DESIGNING AND IMPLEMENTING *CHOICES:* AN OBJECT-ORIENTED SYSTEM IN C++

We describe our experiences in constructing *Choices* and a design methodology that extends existing design approaches by explicitly encouraging specialization, as well as design and code reuse. Although many operating system techniques and designs are well documented, few object-oriented operating systems exist and have been described. SOS [23], CHORUS [18], and Apertos [26] are other examples of object-oriented operating systems.

None of the descriptions of the design of these operating systems provide a methodology for design reuse or describe how the design methodology may be used in conjunction with prototyping. We believe it would be possible to extend our methodology to the design of any of these operating systems. Despite the large number of previous studies and implementations of operating system techniques, there were few guidelines to follow in our project to build *Choices*. Instead, we developed our own methodolgy as the project progressed, adopting many useful ideas from the object-oriented programming and software engineering research communities. We made extensive use of prototyping to guide our efforts, and this approach influenced our concerns for reusing designs. This article is written with the hindsight of building and porting *Choices* to a variety of different hardware platforms. Finally, we discuss the run-time facilities we built to augment the C++ language in order to better support the construction of an object-oriented operating system. These facilities provide garbage collection, first-class classes, dynamically loadable code, and object-oriented debugging.

Existing commercial operating systems provide limited support for software reuse and customized system support for applications. For example, few systems support multiple architectural models for parallelism, message-passing systems, or distributed programming environments. Therefore many programs, including parallel and distributed applications, cannot easily be ported to different machines without substantial modification. An object-oriented operating system encourages reuse by including mechanisms and support for inheritance and specialization. This approach supplements the microkernel, client/server operating system organizations used in systems such as Mach [17], V System [6], and Amoeba [25] to introduce more flexibility for application support. Inheritance encourages the reduction of the implementation of different system services to a small number of classes that can be specialized and combined to achieve a desired result.

*Choices* [5] is an object-oriented operating system that we built using an object-oriented language (C++ [24]). Objects are used to model both the hardware interface, the application interface, and all operating system concepts including system resources, mechanisms, and policies. *Choices* supports an object-oriented application interface based on objects, inheritance, and polymorphism. The application interface is defined by method invocations from objects in user space to objects in kernel space. In user space, a kernel object is represented by an *ObjectProxy* [19]. The objects in *Choices* are organized within a class inheritance hierarchy. The classes are modeled as objects at run time and may be used to browse or examine the contents of an actual *Choices* operating system. User, server, and system objects can be defined, created, and deleted dynamically. The class hierarchies and subsystems we have built are intended to allow further specialization to support a variety of future operating system implementation techniques. Currently, we support several file system formats and interfaces [14], multiple networking strategies including TCP/IP, several message-passing schemes [9], distributed and shared virtual memory [22], and both shared-memory and distributed-memory multiprocessing.

Here, we describe a methodology for designing and implementing an object-oriented operating system in C++. Since C++ supports a minimalist view of object-oriented programming, we extended C++ with run-time support [15, 16] and developed a methodology for reusing design within an operating system [2–4]. The strengths of C++ are that it is efficient [9, 20, 21], portable, and available on a variety of platforms. It is also becoming a popular language in the computing industry. This article differs from an earlier article [4] in that it describes the precise design rules and discusses the integration of our previous methodologies with prototyping.

Our methodology, which is based on frameworks [7], captures design insights using entity relationship diagrams, dataflow diagrams, control flow diagrams, class hierarchies, class interfaces, and path expressions. We will concentrate here on control flow diagrams. Less conventionally, these design properties are inherited from design step to design step, from architectural design to detailed design, and from prototype to final system. We validate the abstract properties of the system by building prototypes that inherit these properties from the abstract design. We customize the inherited design for evaluation in a convenient prototyping environment. Inheritance ensures that our prototype is realistic and consistent with the design for the final system. Ideally, the prototyping environment models as closely as possible the eventual host computer hardware. In practice, the prototype is a specialization of the design that is targeted to run on the Unix operating system in a convenient environment. The prototype allows quick experimentation with design alternatives and encourages reorganization of the class hierarchies of the system to achieve reuse of design, interface specifications, and code. The native implementations of the system are specializations of the design for Sun SPARCstation 1 and 2, Encore Multimax NS32332, IBM AT-compatible Intel 386 and 486 computers, and the Intel Hypercube multicomputer.

Our decision to use C++ was circumscribed by the need to demonstrate an efficient object-oriented operating system. Many of the available object-oriented languages are either interpreted and slow, have considerable run-time support, or include predefined notions of fundamental parts of an operating system such as processes, synchronization, messages, and exceptions. In *Choices,* we have designed and evaluated different implementations of these fundamental concepts. Coding an object-oriented operating system in a nonobject-oriented language has the disadvantages of requiring additional coding effort and reducing the effectiveness of compile-time checking. C++'s strengths are that it lacks complex run-time features, enforces compile-time checking, and compiles into efficient code [21]. Its weakness is that many features of an object-oriented language are either not implemented or are difficult to implement in C++. In *Choices,* we built a class library to augment the language to include first-class Classes, dynamic loading, reference counting, debugging, and proxies—representatives of indirectly accessible objects.

## Methodology

Large and complex software systems are often divided into modules, each module independently designed, and the modules are then assembled into the final system. The decomposition of the system into modules is often called an architectural design for the system.

A framework is an architectural design for an object-oriented system. It describes the component objects of the system and the way they interact. In *Choices,* the components of the system are defined in classes. The interactions among components are defined by constraints, inheritance, inclusion polymorphism (implemented through virtual functions in C++), and rules of composition. *Choices'* frameworks use single inheritance to define class hierarchies and C++ subtyping to express inclusion polymorphism. In practice, we have found the design of a complex system such as an operating system is best organized by a framework that guides the design of subframeworks for subsystems. The subframeworks refine the general operating system framework, as it applies to a specific subsystem.

The framework for *Choices* provides generalized components and constraints to which the specialized subframeworks must conform. It introduces the notion of a *Process* or a sequence of actions, the address space of a *Process* called *Domain,* and the data or *MemoryObjects* that can be accessed by the *Process* in its *Domain* (2). The subframeworks introduce additional components and constraints and subclass some of the components of the framework. Example *Choices* subframeworks include the virtual-memory subsystem, the process management subsystem, the file system, and the message-passing subsystem. Recursively, these subframeworks may be refined further.

Frameworks augment the traditional layered design of operating systems [3]. A layer represents an abstract machine that hides machine dependencies and provides new services. A framework introduces classes of components that encapsulate machine dependencies and define new services. Algorithms or data structures in one layer may be similar to those in other layers, but the layer approach to design has no way to express that similarity. In contrast, a framework may have several different instantiations and implementations within a system; it may be reused.

In our design of *Choices,* a framework for a particular subsystem is characterized by a number of attributes. These attributes are inherited by specializations of the framework. For example, the entity relationships between a process, its domain, and its memory objects are inherited by the three specializations of the *Choices* framework for system, interrupt, and application processes [2]. We have found the following attributes of significance in the design and documentation of an object-oriented operating system: data flow, entity relationships, control flow, path expressions, class hierarchies, and class interfaces [3, 4]. Within a specialization of a framework, the inherited attributes may be extended according to a set of rules ensuring the specialization is consistent with the general design of the framework.

Inheriting the attributes of a framework in its specialization results in large-scale design reuse. Inheritance documents the common attributes of different applications of the same design. Specialization of the different attributes of a framework proceeds in lock-step in our design process. This aids checking the consistency between design steps while a framework is iteratively specialized from a general architectural design into a specific architectural design for a particular subsystem. Finally, objects defined by a framework have properties that can be checked against the attributes of the framework. Our rules for specializing inherited attributes are:

1. An abstract class may be replaced by a concrete class.
2. Abstract and concrete classes may be added.
3. Additional data flows [10], control flows [9], entity relations, and path expressions may be added.

These rules are designed to support substitutability. A specialization may be used wherever a generalization is expected.

Figure 1 shows, as an example, the fault-handling control flow diagram of the *Choices* virtual-memory system. Other attributes, including dataflow, synchronization, and entity relationships, are described in [2], [3], and [4]. The virtual-memory subframework has six components. They are the *Memory Object, Domain, PageFrame Allocator, MemoryObjectCache, AddressTranslation,* and *Disk.* It inherits *Process, Domain,* and *MemoryObject* and their attributes from the *Choices* framework. The *MemoryObjectCache* caches pages of a *MemoryObject* in physical memory. It supervises the transfer of the data in the *MemoryObject* to and from the cache in physical memory. The *PageFrameAllocator* allocates and deallocates pages of physical memory and is used by the *MemoryObjectCaches* to maintain their caches. The *AddressTranslation* encapsulates the address translation hardware of the computer that maps virtual addresses into physical addresses. *Disk* encapsulates a physical storage device like a disk or a RAM disk and is used as backing storage for a *MemoryObject.*

The control flows for both the abstract design and specialization of the virtual-memory system are shown in Figure 1. In this diagram, control is shown flowing between classes of objects. In the actual implementation, the control flow will occur between instances of the classes. An arrow from a source class to a destination class represents a method call from an instance of the source class to an instance of the destination class. A dashed arrow represents a return from a method call. A thick arrow condenses a call to a method and a return from that method. Special entry and exit nodes represent the entry and exit points of the control flow graph. The number labeling an arrow records the ordering of a control flow within a sequence of actions.

The diagram labeled Abstract Control Flow in Figure 1 shows the control flow that is **inherited** by, and therefore common to, all specializations of the virtual-memory subsystem. For example, every instantiation of the framework will invoke the method `addMapping` of an object subclassed from *AddressTranslation* when updating a *MemoryObjectCache.* This flow of control will always occur after the method `read` on the *Disk* returns but before the method `cache` returns from the *MemoryObjectCache.* Specializations of the sybsystem inherit the abstract control flow and augment the design by introducing subclasses. The diagram labeled Virtual Choices Concrete Control Flow shows the inherited virtual-memory subsystem control flow for the prototyping environment. The classes *Disk* and *AddressTranslation* are subclassed to accommodate specializations for the prototyping environment (shown in Figure 1 as *VCDisk* and *VCPageTable*). The *MemoryObjectCache* has been specialized to allow paging using the *PagedMemoryObjectCache.* The control flows between classes are inherited without modification showing a high degree of design reuse between the abstract design and the prototype. This inheritance instills considerable confidence that the prototype represents a consistent implementation of the design. The additional control flows do not reorder or remove the inherited control flows and specialize the abstract design. In our current methodology we have not found the need to remove control flows. If control flows are removed, it makes the reuse of the design much more complex. Our current methodology benefits from the simplicity of the design rules.

Ports of *Choices* to bare hardware exhibit similar inheritance and specializations of the design. Again, the control flows are inherited without substantial additions, and this provides confidence that other *Choices* implementations are consistent with the *VirtualChoices* prototype. The parts of the control flow that are common and have been validated using the *VirtualChoices* prototype should be valid for *Choices* on the bare hardware. For example, in the SPARCstation2 port of the *Choices* operating system, the *Disk* and *AddressTranslation* are replaced by the *SPARCstationDisk* and the *SPARCstationTranslation* concrete subclasses. The resultant concrete control flow diagram is shown in Figure 2.

The control flow diagrams for the SPARCstation2 and *VirtualChoices* ports are quite similar, differing only in two concrete subclasses. Both ports, however, use the same *PagedMemoryObjectCache,* allowing the paging algorithms to be tested on *Virtual Choices* before being tested on the

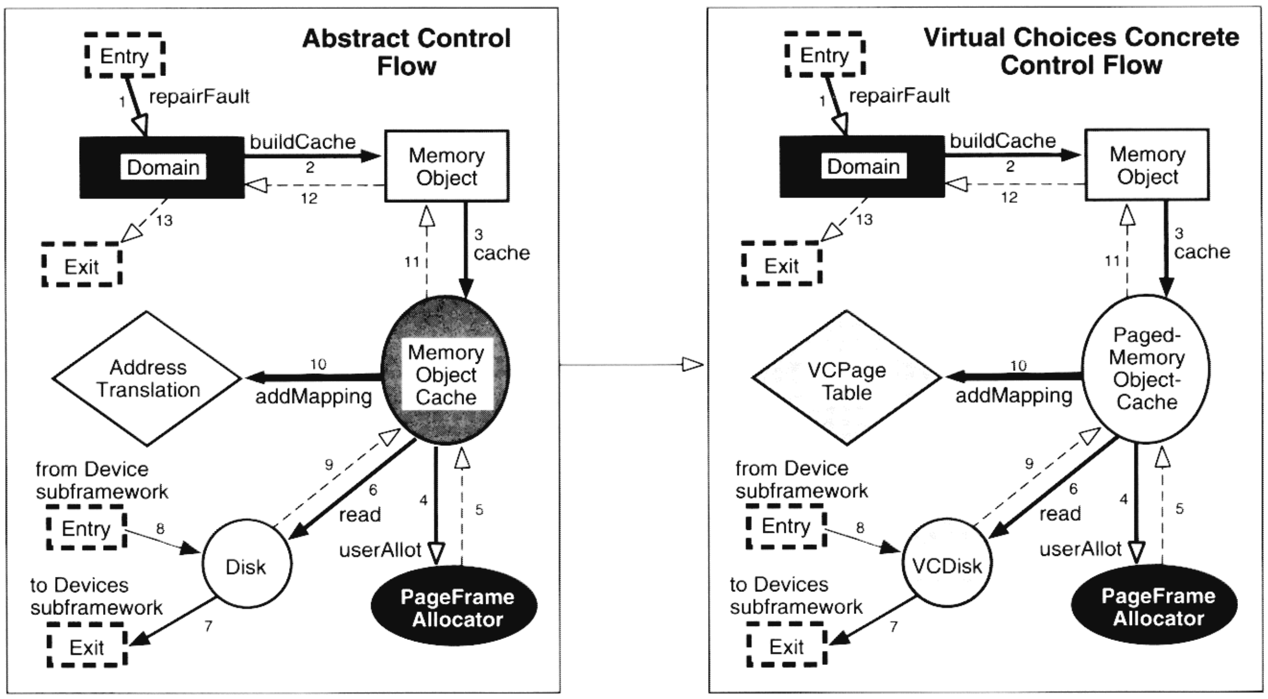


**Figure 1.** Inheriting Control Flow for a page fault: The Abstract Control Flow diagram on the left is inherited by the implementation on the right

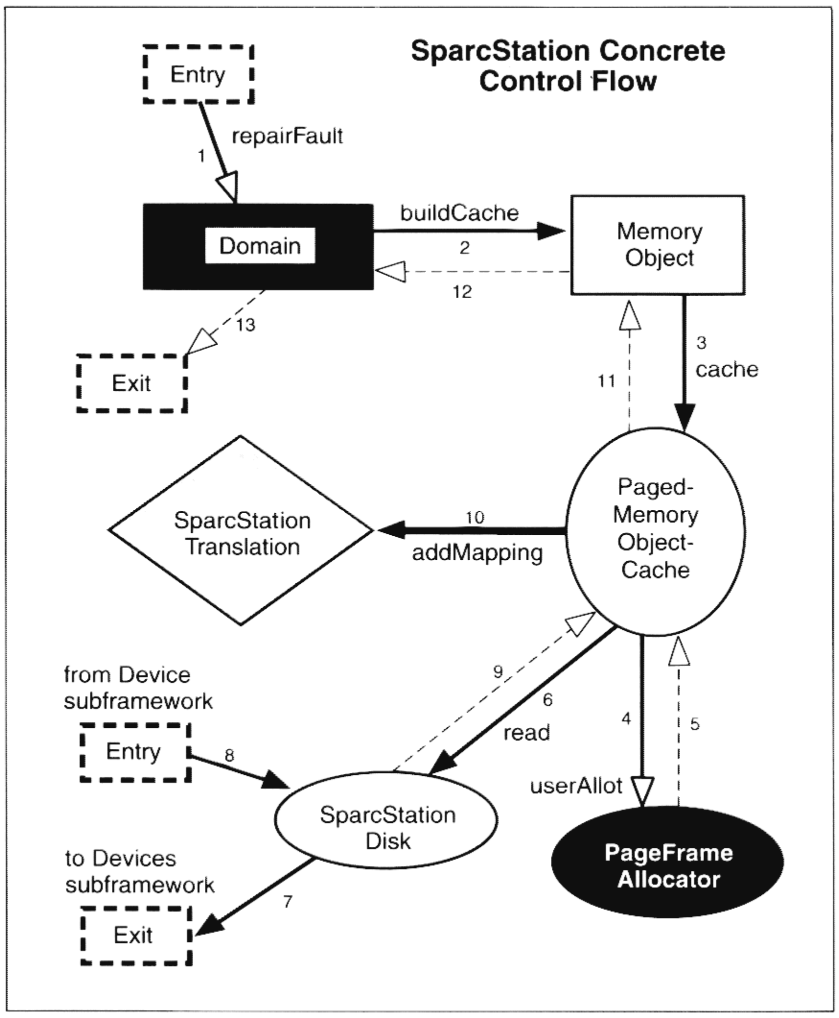


**Figure 2.** The Concrete Control Flow diagram for a page fault on the SPARCstation II

SPARCstation port. This last example stresses the importance of our methodology. The abstract design is maintained between ports.

## Prototyping

*Choices* has a reusable design provided by its frameworks that can be specialized to port *Choices* to different hardware architectures or to prototype *Choices* in a "developer-friendly" environment such as the Unix operating system. We could have used any operating system as the prototyping platform. We chose Unix because it is widely available. Because of the rules for specializing the inherited attributes, there is a large degree of confidence that the prototype and the different ports are consistent in their behaviors.

### Virtual Choices

*VirtualChoices, VC,* is a prototyping tool, and because its behavior accurately reflects the behavior of other implementations of *Choices* on bare hardware, it is an effective port for *Choices* to the "Unix virtual machine."[1] Within the design of *Choices*, code modules representing machine-dependent and processor-dependent algorithms are encapsulated in concrete classes. As an example, in the virtual-memory framework shown in Figure 1, the classes *AddressTranslation* and *Disk* are specialized in implementations with machine-dependent concrete classes.

The *VirtualChoices* implementation specializes the *Choices* architectural design with concrete subclasses (see Figure 1) that implement their functions using Unix system calls. Unlike a simulation [13], *VirtualChoices* behaves like a port of *Choices* to bare hardware. It supports *Choices* applications, the *Choices* trap-based application-kernel interface, virtual memory, paging and page faults, multiple virtual processors, disks, and interrupt-based drivers for the console, timers, and networking. *VirtualChoices* provides a portable, easy-to-use, inexpensive, and tool-rich prototyping environment for *Choices*.

*VirtualChoices* is built using two basic mechanisms: the signal mechanism is programmed to model hardware interrupts for the *Choices* kernel, and the memory-mapped file system is programmed to model *Choices* physical memory and the behavior of a hardware virtual-memory address translation unit.

***Interrupts, exceptions, and the processor.*** In *Choices*, the abstract class *Processor* encapsulates and represents the machine-dependent implementation of interrupt vectors and interrupt handlers. It is designed to map the occurrence of a hardware interrupt or trap into the invocation of a `raise` method on an *Exception* object. An *Exception* object is created for every interrupt or trap in a *Choices* system. The `raise` method handles the condition that caused the trap. *VirtualChoices* uses the Unix process thread of control to execute code, and it uses signals to represent hardware interrupts. *VCProcessor* is a concrete subclass of *Processor* that specializes the abstract *Processor* methods to catch signals representing interrupts and traps. The *VCProcessor* catches signals that are meaningful to *VirtualChoices* by installing an assist function as the handler for signals caused by I/O, page faults, timers, and application traps into the kernel. The assist function maps the signals into corresponding `raise` method calls on *Exceptions*. The *Processor* methods that disable and restore interrupts are implemented in *VCProcessor* by means of the Unix "sigsetmask" call, which blocks and releases signals appropriately.

***Virtual-memory support.*** *VirtualChoices* uses the memory-mapped file system to implement *Choices* physical memory and the behavior of a hardware virtual-memory address translation table. Two regions of Unix virtual memory are reserved for *Choices* application and kernel virtual-memory ranges. The *Choices* physical memory is represented as a Unix file, and logical pages of the file are memory mapped into the *Choices* virtual-memory regions on page boundaries. The *VCPageTable*, a subclass of *AddressTranslation*, manages the mappings using the Unix memory-mapped file system primitives and stores the mappings as an encapsulated data structure to facilitate changing the virtual-memory environment when page tables are switched. Figure 1 shows classes involved in the virtual-memory subsystem of *Choices* and *VirtualChoices*.

The bus and segment signals, SIGBUS and SIGSEGV, occur when *Choices* applications and system programs access virtual-memory addresses that have not been memory mapped into a page of the physical-memory file. The *VCProcessor* implementation maps these signals to the `raise` method on the *Choices VMException*. The *VMException* calls the `repairFault` method on the *Domain* of the current *Process* to handle the fault. The *VirtualChoices* control flow diagram is shown in Figure 1. The control flow sequence for *VirtualChoices* is the inherited Abstract Control Flow with specializations provided by *VirtualChoices* subclasses.

***Devices.*** The *VCDisk* class in *VirtualChoices* specializes a protected *Disk* `doio` method to read and write logical blocks of a Unix file representing a *Choices* disk (see Figure 1). The `doio` method is implemented using the Unix "read" and "write" system calls. In the diagram, the disk is being used as a backing store for the *MemoryObjects* in the *Domain* of a *Choices Process*.

*VirtualChoices* also supports interrupt-driven I/O for the console and Ethernet by catching the SIGIO signal and using nonblocking Unix "read" and "write" calls. The Unix interval timer is programmed to create periodic interrupts, and these are used to drive the timing within *VirtualChoices*.

## The Benefits of Prototyping

*VirtualChoices* provides a "user-friendly" debugging and profiling environment for *Choices* operating system designs. It can be used to prototype *Choices*' machine-independent framework code.

Debugging *VirtualChoices* within the Unix environment is enhanced by a quick reboot/debug cycle time. Further, Unix profiling and debugging tools can be used for design and debugging. Because *VirtualChoices* is a complete implementation of *Choices* that reuses the *Choices* frameworks directly, machine-independent code developed using *VirtualChoices* is directly portable to other *Choices* platforms.

## Run-Time Support for Object-Oriented Systems

In the previous sections, we discussed a design methodology and prototyping environment that proved useful in designing C++ code for the *Choices* operating system. In addition to these approaches, we also found the code development process could be made more productive by augmenting the C++ language with a library that provided facilities more often found in object-oriented languages like Smalltalk. In several cases, the facilities consisted of a set of base classes and a set of programmer conventions that dictated their use. The facilities we found of value are:

- garbage collection

[1] VirtualChoices currently runs on the Sun Microsystems' SUNOS 4.1 version of the Unix operating system.

- the representation of a class by an object at run time
- the dynamic loading of new subclasses and code
- support for debugging classes and instances at run time

These facilities sped debugging and simplified the code of many parts of the operating system, for example, the file system, application interface, name servers, and persistent object code. In this section, we review each of these facilities.

## Garbage Collection

In *Choices,* all resources are represented as objects. In C++, objects must be both constructed and deleted explicitly. This forces the programmer to determine when each object should be deleted. However, objects representing resources in *Choices* may be shared by *Processes* in several *Domains.* It thus becomes very difficult for a programmer to determine when it is safe to delete an object. Automatic deletion of objects when they are no longer required simplifies code and eliminates system programming errors.

Although *ad hoc* implementations of reference counting could be used to determine when to delete objects, the problem arises so often in the implementation of *Choices* that we embedded reference counting within the C++ base classes of the system. We chose reference counting for two reasons: first, reference counting has predictable space and time overheads; and second, we did not want to allow the possible storage leaks that can result from using a conservative garbage collector. Reference counting does not properly handle cyclic data structures. We were able to avoid this problem, however, by identifying all pointer cycles in our design and designating one of the pointers in each cycle as a "weak" pointer. Weak pointers are not counted as references, but they must be maintained carefully to avoid dangling pointers.

***Reference-counting functions.*** A set of classes, methods, and programming conventions was designed to automate, as much as possible, reference counting for object deletion. All objects in the system that require automatic deletion inherit reference-counting behavior from class *Object,* which has an integer reference count and five member functions related to this behavior: `Object()`, `reference()`, `unreference()`, `noRemainingReferences()`, and `~ Object()`.

The constructor for an `Object` initializes its reference count. The public member function `reference` increments the object's reference count; it must be called each time a pointer to an object is stored. The public member function `unreference` decrements the object's reference count and calls `noRemainingReferences` if the object's reference count reaches zero. The `unreference` method must be called each time a pointer to an object is overwritten. The `noRemainingReferences()` method, which should only be called by `unreference`, calls the object's destructor by deleting "this." This method can be overloaded to define other behavior if necessary. The protected, virtual destructor for class *Object* should only be called by `noRemainingReferences`. To avoid premature deletion of objects, all subclasses of class *Object* must define protected destructors.

These five methods provide an effective mechanism that implements reference-counting behavior for objects, but they still place too heavy a burden on the programmer. This burden is the requirement that calls to `reference` and `unreference` functions be placed at *all* appropriate places throughout the code. Experience showed this requirement was too difficult to satisfy. To remove this burden, we chose to treat points to reference-counted objects as objects themselves.

***Pointers as objects.*** The *ObjectStar* class defines "first-class" pointers to objects. Instances of *ObjectStar* have a traditional C++ pointer (`Object * -pointer`) as their only data member. *ObjectStar* defines constructors, destructors, and assignment methods that call the `reference` and `unreference` methods on the object that is pointed to when necessary. *ObjectStar* defines no virtual member functions, thus its instances have no *vtable* pointer and require the same amount of storage as a traditional C++ pointer. We use *ObjectStars* wherever traditional C++ pointers would normally be used for member variables, local variables, global variables, and return values from functions.

## Classes as Objects

The motivation for representing classes as run-time objects arises from several requirements:

- Instances of new subclasses of system abstract classes provide a mechanism to extend the *Choices* application interface in a controlled but nontrivial manner.
- C++ provides safe and efficient compile-time type checking. This type checking can, however, be circumvented if references to objects are passed between *Domains.* Run-time classes allow run-time type checking in such cases.
- Class enquiry functions allow class-based, run-time, controllable debugging, including the ability to list the instances of a class.
- The classes of persistent objects can be recorded as persistent objects.

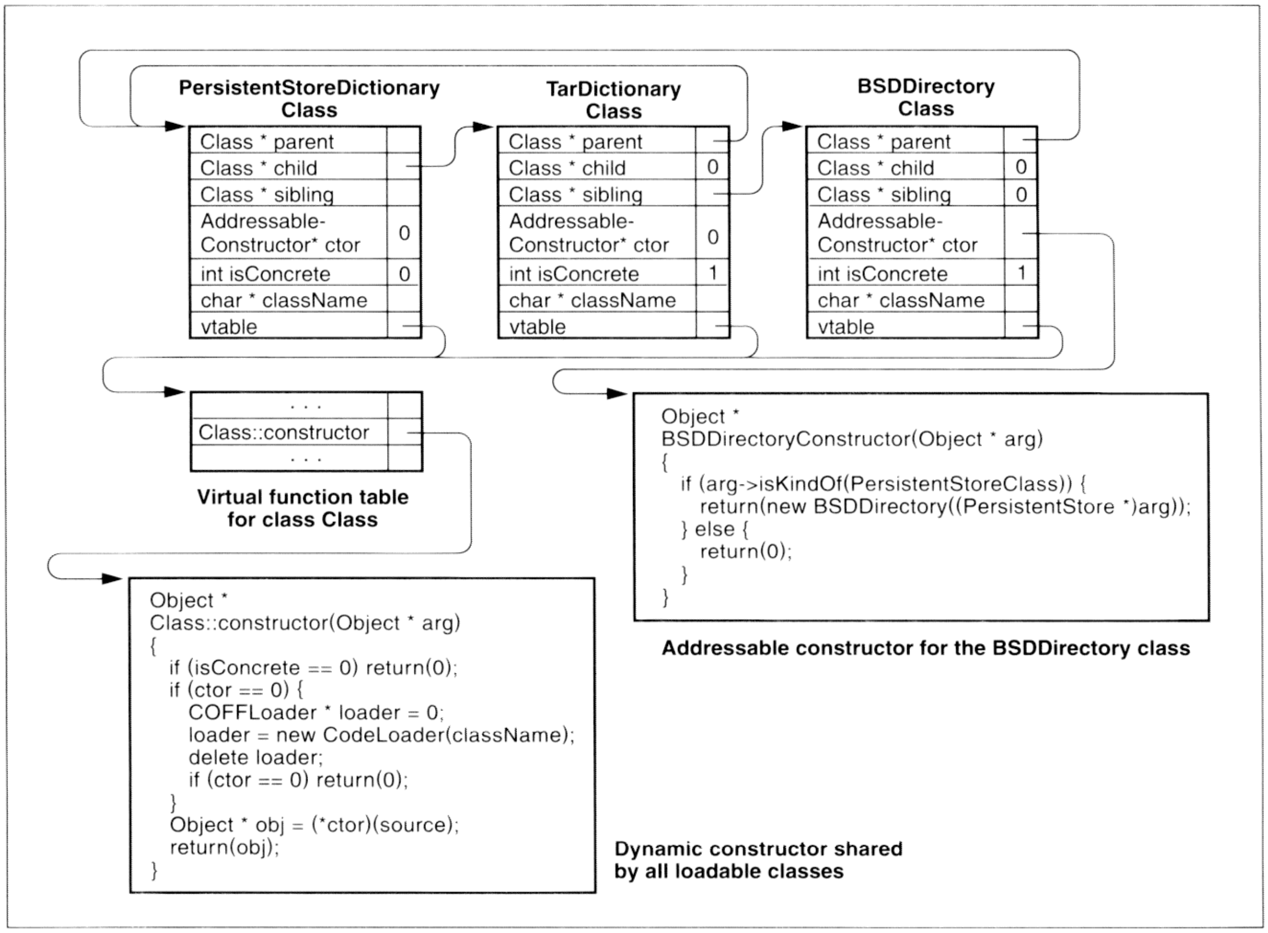
First-class classes or class objects are implemented in *Choices* using a class called *Class;* they are similar to the *Dossiers* described in [8] except that *Classes* also support dynamic code linking and portable debugging.

Users can request information about *Class's* place in the hierarchy. It is searched for by name in the kernel's *NameServer,* and the requested information is displayed. There are three commands for displaying class hierarchy information:

1. ***ancestors,*** which recursively displays the superclasses of the given class
2. ***descendents,*** which recursively displays the subclasses of the given class
3. ***hierarchy,*** which displays first the superclasses and then the subclasses of the class

An example of the ***hierarchy*** command is:

```
Choices> hierarchy UNIXInode
Object
  MemoryObject
    PersistentMemoryObject
      FileObject
```



```
Object *
BSDDirectoryConstructor(Object * arg)
{
  if (arg->isKindOf(PersistentStoreClass)) {
    return(new BSDDirectory((PersistentStore *)arg));
  } else {
    return(0);
  }
}
```

```
Object *
Class::constructor(Object * arg)
{
  if (isConcrete == 0) return(0);
  if (ctor == 0) {
    COFFLoader * loader = 0;
    loader = new CodeLoader(className);
    delete loader;
    if (ctor == 0) return(0);
  }
  Object * obj = (*ctor)(source);
  return(obj);
}
```

**Figure 3.** Dynamic code loading in *Choices*

*Classes* or *Objects*. The *Choices* command interpreter provides an interactive interface to these debugging operations. One can also use a standard debugger such as **gdb** to control the *Choices* object-oriented debugging facility.

The debugging facility is implemented as a set of operations on *Objects* and *Class*, plus statements at the beginning and end of each member function. Operations on objects are divided into four categories: constructors and destructors, reference-counting functions, public member functions, and private member functions. For any class of objects, one can choose to display information about the invocations of any combination of these four types of operations.

## Performance

Object-oriented operating systems can have a comparable performance to existing systems [9, 11, 21]. In this section, we review briefly the performance of our operating system on two architectures: the SPARCstation2 and the Encore Multimax NS32332 shared-memory multiprocessor.

Table 1 shows the context switch times between two application processes. This includes the time to find a new process to run. Next, it shows the time for trapping into the kernel using the *Choices* object-oriented application interface, the *Proxy Call* mechanism. The time for trapping into the kernel and calling a method on a kernel object increases by 5 $\mu$ sec on the SPARCstation2 and by 30 $\mu$ on the Encore Multimax for each additional object passed. Last is shown the round-trip, send-reply time for a 32-byte message sent from process to process over a 10Mb/sec Ethernet interconnecting two *Choices* SPARCstations. This is compared with the time for sending the same message from one application to another on the same *Choices* Encore Multimax shared-memory multiprocessor. Such numbers compare favorably with existing operating systems [1, 25]. Further specializations of *Choices* continue to improve the performance numbers.

## Summary

Since the inception of the project in 1987, we have conducted many experiments demonstrating the benefits, viability, and efficiency of an object-oriented operating system [11, 20]. This article records many of the practical lessons we have learned from this experience, and we hope it will be of value to our colleagues developing the next generation of object-oriented operating systems:

• We chose to implement an object-oriented operating system to help

```
    UNIXInode <<<
      AIXInode
        BSDInode
          SVIDInode
```

There are also two commands for displaying instances of classes:

1. ***members,*** which displays all instances of the given class
2. ***kindred,*** which displays all instances of the given class and its descendants

Examples of these commands follow:

```
Choices> members SVIDInode
SVIDInode [45a180] (/) (:0:2:2)
1 instance.

Choices> kindred UNIXInode
BSDInode [466000] (/) (:0:0:2)
BSDInode   [466100]   (lost+found)
(:0:0:3)
SVIDInode [45a180] (/) (:0:2:2)
3 instances.
```

## Adding Subclasses to a Running C++ Program

*Choices* provides a dynamic loading mechanism that permits applications and system programs to add new system services to the kernel at runtime. A running program needs a dynamic binding mechanism to access newly loaded code. C++ has a dynamic binding mechanism: the virtual function table which is accessible through the object's *vtable* pointer. Once an object's code has been loaded and the object created, its member functions can be invoked through the vtable. However, C++ constructors, which assign vtable pointers to objects at run time, are *statically bound* and cannot be invoked directly from an existing preloaded program. The problem to be solved in an implementation of dynamic, loadable C++ code is to allow existing programs to call the constructors defined in newly loaded classes. To explain our solution to invoking C++ constructors dynamically, we define three kinds of constructor functions: traditional C++ constructors, which have names like Object::Object, addressable constructors that call traditional constructors, which we name in the manner ObjectConstructor, and dynamic constructors used to construct objects using the *Class* hierarchy, which we name Class::constructor.

Traditional constructors cannot be used directly by code that is intended to access the dynamically loaded methods of an object. Instead, the code invokes the dynamic constructor in the *Class* of the dynamically loaded object. When the dynamic constructor operation is invoked on *Class,* it, in turn, invokes the addressable constructor stored in its _constructor instance variable which has been assigned by the loader. The addressable constructor then invokes the traditional constructor. Since the class *Class* is always compiled and loaded with the basic *Choices* kernel, the dynamic constructor can be invoked by the code. The addressable constructor is a function that is compiled at the same time as the traditional constructor. Thus, the addressable constructor can invoke the traditional constructor directly. In this way, the traditional constructor can still be used to allocate and initialize heap storage for the loadable object even though it cannot be accessed directly by existing software.

*Choices* includes a subclass of the abstract class *CodeLoader* for each type of object file format used by the operating system. A *CodeLoader* performs the following operations: locate the symbol table for the running program, locate the code for the classes to be loaded, resolve undefined symbols in the loaded code, relocate symbols in the loaded code, install the address of the addressable constructor in the corresponding *Class* object's -constructor instance variable.

By making C++ classes loadable in *Choices,* parts of the operating system can be greatly simplified. For example, the *Choices* file system supports many types of files. Code for each type of file needs to be loaded only when a request is made to access that type of file. Thus, neither the kernel nor user space is penalized for the flexibility that is provided by the file system. A further benefit of this approach is that it makes loading the class methods for persistent objects simple. Figure 3 shows some of the data structures used to support a dynamic code-loading system in *Choices.* All of the *Classes* in a running *Choices* system form a single tree that corresponds to the compile-time inheritance hierarchy. Figure 3 shows, as an example, three of these *Classes:* the abstract class *PersistentStoreDictionary,* and the concrete classes *TarDictionary*[2] and *BSDDirectory.* All three *Classes* share the same virtual function table and therefore also share the same dynamic constructor. Abstract classes do not need addressable constructors, since programs do not directly create instances of abstract classes; therefore, the *PersistentStoreDictionary* class does not have one. In this example, the addressable constructor for the *BSDDirectory* class has already been loaded, and the *BSDDirectory Class* contains a pointer to it. The addressable constructor for the *TarDictionary* class has not yet been loaded. Any program that tries to access an instance of the *TarDictionary* class by invoking the constructor method on the *TarDictionary Class* will cause a *CodeLoader* to be created. If the *CodeLoader* successfully finds and loads the code for the *TarDictionary* class, it will update the ctor pointer in the *TarDictionary Class.* The constructor method can then call the newly loaded addressable constructor, which in turn will call the traditional constructor for the *TarDictionary* class.

## Portable Debugging

Debugging the kernel of an operating system is difficult, even if it is object oriented. The problem is compounded by a lack of a debugger that can report events in terms of the objects and classes of which the system is composed. Our goal was to develop a flexible debugging environment that was integrated with the *Choices* object-oriented architecture. Therefore, we implemented a debugging facility to selectively display messages on the system console when an operation is invoked on an object.

Debugging messages can be turned on or off at run time, both programmatically and interactively. Applications can control debugging by invoking operations on selected

[2]A TarDictionary makes the names of the files stored in a tar file appear as if they were in a traditional Unix directory.

solve the problem of porting parallel applications to a variety of different computer architectures. The framework and inheritance methodology we developed allowed us to organize the design and code of the resulting implementations.

• We selected C++ as an efficient, viable implementation language for object-oriented operating systems. The language has a minimal number of object-oriented features. As we developed *Choices*, we began to appreciate many of the high-level features that are available in other less efficient object-oriented languages. Most of the facilities we desired could be programmed within C++ without extending the language. This suggests that an object-oriented language for systems programming needs a small set of appropriate object-oriented features.

• C++ does not have concurrent-programming features. We were able to build such features using C++ language primitives, classes, and subclasses. For our work, the lack of these C++ features was an advantage, since it allowed us to develop operating system implementations of concurrent programming that were object-oriented and benefited from inheritance and polymorphism.

• In developing libraries to support missing object-oriented language facilities and concurrency, the design methodology helped organize the resulting code.

• The design methodology serves as excellent documentation for the design of *Choices*.

• Progressing through a series of experiments with customized file systems, message-passing systems, and virtual memory, we generalized the class hierarchies of *Choices* to provide a more structured organization to our system. Building useful generalizations and class hierarchies is different from stepwise refinement of a single solution. It is, instead, a way of organizing conclusions gathered from iterative design and prototyping.

• The object-oriented encapsulation of the hardware in our system allowed us to use inheritance and specialization to prototype the system in an easy-to-use software development environment and port the system to various hardware platforms.

In conclusion, we advocate the design of an object-oriented operating system using frameworks as an effective software engineering technique, both to increase productivity and to enhance one's ability to build innovative new systems through design and code reuse. Using the frameworks as a tool provides high-level architectural-design reuse. Finally, prototyping is a powerful tool when coupled with object-oriented design and frameworks. **C**

**Table 1.** The performance of the basic *Choices* operating system primitives

| Performance of *Choices* (in μsec) | | |
|---|---|---|
| **OS Primitive** | **SPARCstation II** | **Encore Multimax** |
| Context switching | 150 | 412 |
| Proxy call | 38 | 72 |
| Message passing | 1,800 | 700 |

**References**

1. Anderson, T.E., Levy, H.M., Bershad, B.N. and Lazowska, E.D. The interaction of architecture and operating system design. In *ASPLOS, The International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr. 1991), pp. 108–120.
2. Campbell, R. and Islam, N. *Choices:* A parallel object-oriented operating system. *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, Cambridge, Mass., 1993. To be published.
3. Campbell, R.H. and Islam, N. A technique for documenting the framework of an object-oriented system. In *Proceedings of the Second International Workshop on Object-Orientation in Operating Systems* (Paris, France, Sept. 1992).
4. Campbell, R.H., Islam, N. and Madany, P. *Choices,* frameworks and refinement. *Comput. Syst. 5,* 3 (1992).
5. Campbell, R.H., Russo, V. and Johnston, G. Choices: The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop* (Santa Fe, New Mex., Nov.). USENIX Association, 1987, pp. 109–123.
6. Cheriton, D. The V distributed system. *Commun. ACM 31,* 3 (Mar. 1988), 314–334.
7. Deutsch, L.P. Design reuse and frameworks in the Smalltalk-80 programming system. In *Software Reusability.* V 2. ACM Press, New York, 1989, pp. 55–71.
8. Interrante, J.A. and Linton, M.A. Run-time access to type information in C++. In *Proceedings of the USENIX C++ Conference* (San Francisco, Calif., Apr.). USENIX Association, 1990, pp. 233–240.
9. Islam, N. and Campbell, R.H. Design considerations for shared memory multiprocessor message systems. *IEEE Trans. Parall. Distrib. Syst.* (Nov. 1992), 702–711.
10. Islam, N. and Campbell, R.H. Reusable dataflow diagrams. Tech. Rep. UIUCDCS-R-92, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Ill., 1992.
11. Islam, N. and Campbell, R.H. Uniform coscheduling using object-oriented design techniques. In *Proceedings of the International Conference on Decentralised and Distributed Systems* (Palma, Spain, Sept. 1993).
12. Islam, N., Mcgrath, R.E. and Campbell, R.H. Parallel distributed application performance and message passing: A case study. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems.* (San Diego, Calif. Sept. 1993).
13. Johnston, G. and Campbell, R.H. A multiprocessor operating system simulator. Tech. Rep. UIUCDCS-R-88-1460, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Ill., 1988.
14. Madany, P.W. *An object-oriented framework for file systems.* Ph.D. dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Ill., 1992.
15. Madany, P.W., Campbell, R.H. and Kougiouris, P. Experiences building an object-oriented system in C++. In

*Technology of Object-Oriented Languages and Systems Conference* (Paris, France, Mar. 1991).

**16.** Madany, P., Islam, N., Kougiouris, P. and Campbell, R.H. Practical examples of reification and reflection in C++. In the *International Workshop on Reflection and MetaLevel Architecture* (Nov. 1992), pp. 76–81.

**17.** Rashid, R. Threads of a new system. *UNIX Rev.* (1986).

**18.** Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Lanlois, S., Leònard, P. and Neuhauser, W. CHORUS distributed operating systems. *Comput Syst. 1,* 4 (1988).

**19.** Russo, V.F. An object-oriented operating system. Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, Urbana, Ill., 1991.

**20.** Russo, V.F., and Kaplan, S.M. A C++ interpreter for Scheme. In *Proceedings of the USENIX C++ Conference* (Denver, Col., Oct.). USENIX Association, 1988, pp. 95–108.

**21.** Russo, V.F., Madany, P.W. and Campbell, R.H. C++ and operating systems performance: A case study. In *Proceedings of the USENIX C++ Conference* (San Francisco, Calif., Apr.). USENIX Association, 1990, pp. 103–114.

**22.** Sane, A., MacGregor, K. and Campbell, R. Distributed virtual memory consistency protocols: Design and performance. In the *Second IEEE Workshop on Experimental Distributed Systems.* IEEE, New York, 1990.

**23.** Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M. and Valot, C. SOS: An object-oriented operating system—Assessment and perspectives. *Comput. Syst. 2,* 4 (1989).

**24.** Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1991.

**25.** Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G.J., Mullender, S.J., Jansen, J. and Van Rossum, G. Experiences with the Amoeba distributed operating system. *Commun. ACM 33,* 12 (Dec. 1990).

**CR Categories and Subject Descriptors:** D.1.3 [**Software**]: Programming Techniques—*concurrent programming;* D.2.2 [**Software Engineering**]: Tools and techniques—*software libraries;* H.2.4 [**Database Management**]: Systems—*concurrency*

**General Terms:** Design, Performance

**Additional Key Words and Phrases:** Concurrency, concurrent object-oriented programming

**About the Authors:**

**ROY H. CAMPBELL** is a professor in the department of Computer Science at the University of Illinois, Urbana-Champaign. Current research interests include distributed and parallel systems, operating systems, programming language design, testing, and software engineering.

**NAYEEM ISLAM** is a Ph.D. candidate in computer science at the University of Illinois Urbana-Champaign. Current research interests include parallel and distributed operating systems and object-oriented programming.

**DAVID RAILA** is a research staff programmer in the Computer Science department at the University of Illinois at Urbana-Champaign. Current research interests include operating systems, networking, and high-performance parallel computing.

**Authors' Present Address:** University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 West Springfield Avenue, Urbana, IL 61801; email: roy, nislam, raila@cs.uiuc.edu

**PETER MADANY** is staff engineer at Sun Microsystems. Current research interests include object-oriented programming, file systems, and distributed operating systems. **Author's Present Address:** Sun Microsystems Laboratories, 2550 Garcia Avenue, MTV29-112, Mountain View, CA 94043; email: madany@eng.sun.com

This work was supported in part by NSF grant CISE-1-5-30035, NASA grant NAG 1-163, IBM agreement MHVV2431, and Unix System Laboratories. Nayeem Islam is supported in part by an IBM Graduate Fellowship.