Considerations of Persistence and Security in Choices, an Object-Oriented Operating System*

Roy H. Campbell† and Peter W. Madany
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801

Abstract

Choices is an object-oriented operating system written in an object-oriented language and runs on bare hardware. It supports distributed, parallel applications on a network of multiprocessors. The kernel is implemented as a dynamic collection of objects that have been instantiated from classes. The classes are represented as objects at run-time. *Choices* has a paged virtual memory organized around memory objects. Each memory object can have its own separate backing store, page placement, and page replacement algorithms. It can be shared, both within a shared memory multiprocessor and between networked computers using a distributed virtual memory protocol.

Choices supports an object-oriented file system model in which files may be mapped into virtual memory. Together, the kernel and an object file store specialization of the file system model provide persistent objects. Applications are permitted controlled access to the methods of persistent objects. Security and protection are provided by a combination of object proxies, access lists, and name servers. This paper discusses the persistence and security design issues that are being studied in the *Choices* implementation.

1 Introduction

Choices is an object-oriented operating system written in an object-oriented language (C++). The system runs on bare hardware: the NS32332, the MC86030, and the Intel 386. It supports distributed, parallel applications on a network of Encore Multimax multiprocessors.

Choices has, as its kernel, a dynamic collection of objects. System resources, mechanisms, and policies are represented as objects that belong to a class hierarchy.[1] For programming convenience, the root of the hierarchy is an *Object* and the classes of the hierarchy are represented as *Class* Objects. A Class supports several methods including:

- isMemberOf, which takes a Class as its argument and returns whether the Object is an instance of the given Class,
- isKindOf, which takes a Class as its argument and returns whether the Object is an instance of the given Class or any of its subclasses, and
- isASubclassOf, which takes a Class as its argument and returns whether the Class is a subclass of the given Class.

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

†Professor Campbell is on sabbatical at the University of Edinburgh.

[1] By convention, we use an initial capital letter to designate the class of such objects. Where we unambiguously refer to an instance of a class, we will use the name of the class of the object.

All entities in the operating system are modeled as objects and include system processes, user processes, files, regions of memory, and hardware devices like CPU's and disk controllers. The application/kernel interface is defined by method invocations from objects in user mode to objects in kernel mode. In user mode, a kernel object is represented by an *ObjectProxy*. An ObjectProxy may be static or created dynamically on demand if the protection policy of the kernel object is not violated. Kernel objects are mapped into ObjectProxies by special kernel objects called *NameServers*. The performance of the ObjectProxy approach is described in Section 7.

A *Choices* Process executes in a Domain, which maps a collection of logical data segments or MemoryObjects into a virtual memory. Each MemoryObject can be paged to its own backing store using independent paging algorithms. Distributed virtual memory[6] allows the dynamic addition or removal of a MemoryObject to or from the different virtual memories of a network of computers.

Persistent storage is based on an object-oriented file system model. MemoryObject subclasses support access to data stored on disks, files, or physical memory in a variety of formats. The formats include interface and data representations for UNIX 4.3 BSD file systems, MS-DOS file systems, and UNIX System V file systems [9, 8]. Files may be mapped into virtual memory and this allows transparent access to persistent data. Distributed virtual memory will allow files to be accessed remotely.

An object store is a specialized application of the file system. PersistentObjects reside in the store as MemoryObjects and are mapped into virtual memory on request. Method invocations on PersistentObjects are identical to method invocations on normal Objects. Distributed virtual memory will allow remote method invocations on PersistentObjects.

Security is provided by a combination of access rights maintained by the file system, individual objects, and a user/system protection mechanism. Kernel objects are protected by supervisor state and by the virtual memory hardware. Method calls to kernel objects from user applications cannot be performed unless the user has been granted permission to access the object. That permission is provided in the form of an ObjectProxy. Once provided, all method calls to the ObjectProxy are trapped and converted to method calls to the kernel object. During the trap, they are checked for validity by a kernel protection mechanism. Non-kernel objects are mapped into the virtual memory space of the application. Their data segments may be shared between applications using shared virtual memory or distributed virtual memory. Once access is established, method calls can be made directly from the application to the object.

This paper discusses the persistence and security design issues that are being studied by the Choices implementation.

2 Virtual Memory support for MemoryObjects

A *Choices* process executes in a Domain, which is a mapping between a virtual memory space and MemoryObjects[14, 16]. Each memory-mapped MemoryObject has a single MemoryObjectCache that maintains physical memory management information associated with virtual memory. The information is kept in a machine independent and virtual memory address independent form. This allows a Domain to map a particular MemoryObject into several regions of its virtual address space. It also allows several Domains to share both a MemoryObject and its MemoryObjectCache and common physical memory management information.

The Domain resolves the virtual memory address of a page fault into a MemoryObject and offset pair. The Domain requests the MemoryObject to map the offset. In turn, the MemoryObject requests its MemoryObjectCache to repair the fault. The paging strategy is encapsulated entirely within the MemoryObjectCache. If required, a page frame is requested from the frame Store. The Store returns a descriptor that identifies the physical memory into which data will be paged. The MemoryObject fetches the data from backing store and returns the descriptor to the Domain. In turn, the Domain passes the descriptor to an AddressTranslation method that updates the hardware virtual memory mapping. Then the process that generated the page fault is resumed.

Distributed virtual memory is implemented by a subclass of MemoryObjectCache[6, 16]. Page coherency

is maintained through DVMPageRecords. Each DVMPageRecord defines the state of a page. The coherency protocol is defined by a state machine and state transitions. The current protocol used allows one computer to own a page at a time and ownership is driven by a demand to write. The owner responds to remote read requests for the page by changing its page to read-only access and copying the page to the requesting computer with read-only access. Before a write, all remote read-only copies of the page are invalidated.

Since distributed virtual memory is associated with the MemoryObject, the scheme is flexible and permits a Domain to share many distributed MemoryObjects, each of which may use a different coherency protocol and be shared by a disjoint set of remote computers. Distributed virtual memory is established by mapping a remote MemoryObject into virtual memory. A remote MemoryObject can be obtained in many ways including through method invocation on PersistentObjects and through the distributed file system interface.

3 An Object-Oriented File System Model

The object-oriented file system model[7] structures the data of MemoryObjects with *StoredObject*s. The two main subclasses of StoredObject that organize storage are ObjectContainer and ObjectDictionary. FileStream and PersistentObject are two further subclasses of StoredObject that provide useful programming abstractions of a MemoryObject.

A MemoryObject provides read and write methods to access the logical collection of persistent data that it manages. The read and write methods transfer *blocks* of data. Additional methods are used to support memory-mapping and protection. MemoryObject subclasses are specialized by the storage mechanism used to store the data and include Disks that read and write sectors, Partitions that read and write clusters of disk sectors within a contiguous region of a Disk, and various logical "disks", or files, like UNIX inodes. Along with its data, the MemoryObject also records a specific subclass of StoredObject, and this determines how its data may be used. Each StoredObject is *based on* a single MemoryObject, which is called its *underlying* MemoryObject.

An ObjectContainer organizes a MemoryObject into an indexed collection of MemoryObjects. Compared with the original MemoryObject, the MemoryObjects of a Container correspond to a more abstract storage mechanism or higher level within the file system. The Container may impose a dynamic or static organization on the MemoryObject depending on the abstraction being implemented. By definition, MemoryObjects must belong to exactly one ObjectContainer and can be subdivided into at most one ObjectContainer. ObjectContainers can be nested to an arbitrary depth. A MemoryObject descriptor called an *IdNumber* includes a list of indices that uniquely identify the location of a MemoryObject stored in nested ObjectContainers.

All StoredObjects and their underlying MemoryObjects are persistent objects that can be activated and deactivated as well as created and deleted. Except for PersistentObjects (see Section 5), the activation and deactivation of StoredObjects is explicitly programmed within the methods of the Objects. ObjectContainers provide the methods open, close, and create to activate, deactivate, and create MemoryObjects, respectively. MemoryObjects are deleted if, after they have been closed, they are no longer referenced by any other objects in the file system. The open method takes an index as argument and returns the corresponding MemoryObject. Internally, the ObjectContainer associates a descriptor with each index that includes a reference to the MemoryObject once it has been opened. Successive opens return the reference obtained by the first open. An *activation* ReferenceCount is maintained by opens and closes and when this count returns to zero, the MemoryObject is deactivated.

ObjectContainer subclasses are specialized by the scheme used to provide an indexed collection of MemoryObjects and include DiskContainers, which divide the storage of a Disk into an collection of Partitions, and various stream-oriented file system containers, which divide the storage of a Partition into a collection of files and free blocks.

An ObjectDictionary, also called a directory, uses its underlying MemoryObject to store a mapping from convenient symbolic keys for MemoryObjects to the indices used by an ObjectContainer. Within any ObjectDictionary, the keys must be unique, but several keys may map to the same index. MemoryObjects can belong to one or more ObjectDictionaries. The open method takes a key as an argument and, if the key

is found, returns the appropriate MemoryObject. It obtains this reference by invoking the open method on the ObjectContainer using the appropriate index. ObjectDictionaries also have methods to add and remove mapping entries. When a MemoryObject is added to an ObjectDictionary, the MemoryObject's *link* count is incremented. When a MemoryObject is removed from an ObjectDictionary, the MemoryObject's link count is decremented. ObjectContainers can use this link count to determine when MemoryObjects are no longer needed and therefore need to be deleted. ObjectDictionary subclasses are specialized by various file system standards for describing the storage layout of mappings.

The StoredObject subclass, FileStream, provides applications with a stream-oriented interface to MemoryObjects. FileStreams provide byte-addressability and the concept of a "current file position". Applications may use read and write methods to read and write multiple bytes of sequential data. These methods change the current file position. The seek method also changes the current file position. A FileStream either buffers individual application reads and writes and invokes block reads and writes on its associated MemoryObject, or it directly exploits the memory-mapping of MemoryObjects into an application's Domain.

Specializations of the file system model provide stream-oriented file systems that conform to operating system standards such as 4.3 BSD UNIX, System V UNIX, and MS-DOS[9, 8]. Preliminary performance data has been gathered for the 4.3 BSD UNIX specialization, see Section 7. The file system class hierarchy also supports the construction of customized and experimental file systems. Various instances of file systems can coexist and interoperate in a running *Choices* system.

The experimental file systems that have been built include a log-structured file system described in [11]. Log-structured file systems are designed to increase I/O throughput by reducing disk head movement. We have two working prototypes of log-structured file systems and we plan to measure and analyze their performance. Another experimental file system provides the UNIX stream-oriented interface to formatted files of various kinds including libraries and archives. The *Choices* stream-oriented file system tools can then be used on these formatted files without modification. A file system for the UNIX "ar" format has been finished and "tar", "cpio", "a.out", and "Mail" formats are in progress.

4 An Object-Oriented Kernel Interface

The *Choices* kernel is a dynamic collection of objects that is structured by a class hierarchy. Applications access kernel facilities by invoking methods on kernel objects. Abstract classes provide interfaces for generic services within the kernel, for example I/O services. New objects can be installed within the kernel and their methods can be invoked both by application programs and by other system objects using the abstract interfaces.

The ObjectProxy class provides a transparent capability to invoke the methods of kernel objects. An ObjectProxy is a protected object that delegates method calls to a specified object. The use of ObjectProxies requires no compiler modifications. They are syntactically and semantically identical to the objects they represent and may be used interchangeably with them. Kernel objects may be used instead of application objects in a user program with no change to the program. Kernel objects may also use ObjectProxies to access other kernel objects.

Without an appropriate ObjectProxy, application program access to kernel objects is prevented by the virtual memory mapping and supervisor mode protection of the hardware. An ObjectProxy is allocated in read-only memory so that it may be modified only by trusted system code. ObjectProxies are obtained from a NameServer that resolves requests involving symbolic names into ObjectProxy references.

An application invoking a method call[2] on an ObjectProxy follows the standard C++ conventions for method lookup by jumping indirectly through an indexed address in the virtual function table (vtable) associated with the object. The first word of a C++ object contains the address of its vtable. The second word of an ObjectProxy contains the address of the actual object to be accessed. All ObjectProxies share a common vtable and the resulting code that is executed records the method index and reinvokes the method on the actual object. When an application is making the method call, the code traps into kernel code before

[2]For an object to be *proxiable*, its public methods must be *virtual* functions.

completing the method call reinvocation. Care is taken in the code to check that an ObjectProxy is not located in application memory.

5 Persistent Objects

In *Choices*, the concept of a *persistent object* provides a unification and simplification of several subsystems. A persistent object is a member of a subclass of PersistentObject, which has methods, local data, and a lifetime comparable to a file in a file system. Its lifetime is "global"; that is, it does not depend on the lifetime of application or system processes and can survive reboots of the system. However, its lifetime does depend on maintaining the integrity of the *Choices* file system. To simplify programming, PersistentObjects may store references to other PersistentObjects in their local data. To avoid making a distinction between a PersistentObject and a conventional object, activating and deactivating a PersistentObject is performed transparently. This is in direct contrast with the way that persistent data is accessed in the file system, where files must be opened and closed explicitly.

5.1 Persistence

The mechanisms to support PersistentObjects are built as a specialization of existing mechanisms: the *Choices* object-oriented file system model and memory-mapped files. Except for the following minor restrictions, a PersistentObject has transparent usage:

1. A pointer cannot be used to store the virtual memory address of a PersistentObject within the local data of a PersistentObject because that address may be invalid or inappropriate in one of the many different virtual memory spaces into which the object may be mapped during its lifetime. Instead, PersistentObjects must use a *Reference* to store a descriptor for another PersistentObject. A Reference is a lightweight object that has the same operational syntax as a pointer.

2. The creation of a PersistentObject uses a method call rather than the standard C++ new operator.

These restrictions arise because *Choices* currently runs on 32-bit address virtual memory architectures. To maximize the virtual memory space available to a process, no virtual address range is used by deactivated PersistentObjects. PersistentObjects may be mapped at different addresses within different Domains during their lifetime. The virtual memory address of a PersistentObject within a particular Domain is always the same during an activation although it might change after a period of deactivation. The Reference mechanism allows the operating system to avoid potential address conflicts.[3]

References contain a "pointer" that locates a PersistentObject within a persistent *object store*. When a Reference is first assigned the location of a PersistentObject, the corresponding MemoryObject may not be memory-mapped into the Domain containing the Reference. Before the Reference can be dereferenced to invoke a method on the PersistentObject, the PersistentObject must be "active" and *bound* to an address within the virtual memory. The initial dereference binds the Reference to an address and stores the address for future use. Subsequent method calls dereferencing the Reference use the stored address. The following steps are performed each time a Reference is used:

1. Check if the PersistentObject has been bound to a virtual memory address. If it has, proceed to step 10.

2. Request the retrieval of the underlying MemoryObject of the PersistentObject from the file system.

3. If the MemoryObject has not already been opened, the file system opens it.

[3] A port of *Choices* to a machine with a 64- or 128-bit virtual memory address space would allow a single large virtual memory address space for all Domains. PersistentObjects could be assigned an address for the duration of their lifetime. This would allow us to remove the restrictions but would not require a change to applications or PersistentObject code.

4. Check the class of the MemoryObject against the class of the Reference.
5. If the code for the methods of the PersistentObject has not already been loaded, load the code from the file system.
6. If the MemoryObject has not already been mapped into any Domain, create a cache for it.
7. Set the vtable pointer for the PersistentObject to the correct value.
8. If the MemoryObject has not already been mapped into the current Domain, map the MemoryObject-Cache into the current Domain.
9. Store the virtual memory address of the mapped MemoryObject for future use.
10. Proceed with the method call using the virtual memory address.

If the PersistentObject is a kernel object and the Reference is stored in the virtual memory of an application, the Reference will, if permitted by the NameServer, bind the address to an ObjectProxy representing the object. Having introduced the PersistentObject scheme, several Reference mechanism design considerations require further explanation.

5.2 References

References, which are used instead of pointers in the local data of PersistentObjects and applications, allow semi-permanent storage of complex data structures. References are only used to refer to PersistentObjects, thus the PersistentObject implementation imposes neither a time nor a space overhead on the use of memory-only C++ objects. They provide the following features: compile-time type checking, a syntax identical to pointers, transparent object activation and deactivation, and automatic garbage collection.

Type Checking To help ensure the correct usage of PersistentObjects, References are typed by a Reference class hierarchy that mirrors the PersistentObject class hierarchy. Typed References allow almost all uses of PersistentObjects to be type-checked at compile-time. (C++ compilers usually perform most type-checking for C++ programs.) Run-time type checks are needed only when the file system interface retrieves a PersistentObject from the file system's object store.

Pointer-like Syntax A Reference encapsulates storage for the IdNumber of a PersistentObject's underlying MemoryObject. The IdNumber is sufficient information to activate the PersistentObject. A Reference overloads the C++ assignment operator "=" with an assignment method. The IdNumber in a Reference may be changed by assignment using another Reference as an argument. For example, file system interface enquiry methods use assignment to return a Reference. The assignment method maintains a reference count of how many References contain IdNumbers for a particular PersistentObject.

The C++ dereferencing operator "->" is overloaded so that References may be used as pointers without any syntactic changes to the code. The first time a method on a Referenced PersistentObject is invoked within a Domain, its underlying MemoryObject must be mapped into virtual memory and it may need to be activated. These steps are performed by the dereferencing method.

The implementation maintains a read-only persistent object hash table of IdNumbers in the virtual memory of each Domain. The hash table associates a pointer with the IdNumber. The dereferencing method hashes the IdNumber into the table. The first time this happens, the IdNumber will not be found in the table. The dereferencing method uses the IdNumber to map the PersistentObject representation into virtual memory using the file system interface. During this update, the pointer associated with the IdNumber in the hash table is assigned the virtual address used by the Domain to map the MemoryObject into virtual memory. Subsequent dereferences hash into the table and use the virtual memory address that is stored in the associated pointer.

Activation and Deactivation The number of References "pointing" to a PersistentObject per Domain is recorded in an *activation* ReferenceCount. This ReferenceCount is increased whenever a Reference "pointing" to the PersistentObject is copied by assignment into another Reference.

PersistentObjects are retrieved from the object store, activated, and mapped into the virtual memory of a Domain by an initial Reference dereference. Further dereferences may map a PersistentObject into other Domains. The PersistentObject's MemoryObject maintains a *domain* ReferenceCount that contains the number of Domains into which it has been mapped. It is incremented when the MemoryObject is mapped into a new Domain.

Whenever a Reference is deactivated, reassigned, or destroyed, the activation ReferenceCount of the PersistentObject to which it "pointed" is decremented. If it reaches zero, the PersistentObject is removed from the virtual memory of the Domain, the persistent object hash table entry is removed, and the domain ReferenceCount is decremented. If the domain ReferenceCount reaches zero, the PersistentObject is deactivated and its data stored in the file system.

Garbage Collection To conserve file space in a persistent environment, References help support "automatic" garbage collection of PersistentObjects that are no longer needed. This garbage collection is based on the concept of *unreachable* objects. The persistent object store specialization of the model file system contains a PersistentObject "forest" directory that is, by definition, reachable. The directory contains References to reachable PersistentObjects. In turn, these PersistentObjects may have References to other PersistentObjects and so on, forming a chain of References. A reachable PersistentObject must be in a chain of References from the forest. All other PersistentObjects are considered unreachable and their storage will be automatically reclaimed. A PersistentObject can be added to or removed from the forest using the methods persist and desist.

To simplify the current implementation, *link* counts are used to find unreachable PersistentObjects. The management of link counts is transparent to programs that use PersistentObjects, and is similar to the reference counting mechanism used for activation and deactivation. When a Reference is assigned a PersistentObject's IdNumber, the PersistentObject's link count is incremented. If the Reference is destroyed or has another IdNumber assigned to it, the previous PersistentObject's link count is decremented. When a link count reaches zero, the PersistentObject is considered unreachable, and it is deleted.

The current object store supports garbage collection for frameworks of PersistentObjects that are modeled as directed acyclic graphs (DAG's). Garbage collection of cyclical data structures is planned but not currently supported.

Because one of the first applications of the *Choices* object store, an object-oriented software configuration management system[12], requires a framework modeled as a DAG with back-edges, two classes of References are supported in the current experimental implementation of the object store: References and TransientReferences.

References model forward-links, also called "strong" pointers. The ReferenceCounts control activation and deactivation. File system link counts control garbage collection and keep needed PersistentObjects from being deleted. TransientReferences model backward-links, also called "weak" pointers. They use ReferenceCounts only. Thus, they do not keep PersistentObjects from being deleted.

TransientReferences are used to point from component objects back to aggregate objects that Reference them. Reference cycles would occur if the component objects also used References, preventing our simple garbage collection scheme from working. Implementation of improved garbage collection methods will eventually make these subclasses of Reference unnecessary.

Distributed and Shared Persistent Objects A PersistentObject may be shared between Domains. A MemoryObjectCache, together with any mutual-exclusion policy implemented by subclasses of the PersistentObject class, will keep the data representation of the PersistentObject coherent. When the PersistentObject is no longer needed in virtual memory, any dirty pages are paged back to the file system and its MemoryObject is closed.

The concept of a PersistentObject extends to a distributed system using distributed virtual memory. The IdNumber used in a Reference to activate a PersistentObject is designed to locate a MemoryObject in a "distributed" file system. Thus, remote References may be assigned to local References and dereferenced to invoke methods on possibly remote PersistentObjects.

6 Issues of Security

Our goal is to recast operating system security and protection issues into an object-oriented framework. Security and protection policies and mechanisms should be encapsulated within objects. Related policies and mechanisms should be organized within a class hierarchy that allows the reuse of code. In this paper, we use security in the context of user concerns and protection in the context of the operating system.

In *Choices*, protection is physically implemented by the virtual memory hardware and user/supervisor state and these are under the control of the kernel. Applications can only access kernel objects if the kernel NameServer builds an ObjectProxy to allow that access. Two important objects that the NameServer provides the application are the persistent object store and file system. The object store controls access to PersistentObjects. The file system controls access to persistent data. More complex security and protection mechanisms and policies are built from these primitives.

Naming In *Choices*, we have tried to separate the issues of naming from protection. There are several different kinds of names that designate objects including:

1. a virtual memory address,
2. a MemoryObject IdNumber, or
3. a symbolic "path name" of keys that identify a file or PersistentObject through a sequence of directory searches.

A program may have access to a name but may not be able to invoke a method on the object that it designates. In general, permission to access the methods of a named object must be granted by an object implementing a protection or security policy before the named object can be used. Protection and security mechanisms ensure that an object cannot be used until this has been granted by a policy. In our study of protection and security so far, we believe that we can restrict the acquisition of names in "need to know" protection policies by protecting NameServers from general access.

ObjectProxies ObjectProxies are like capabilities in the sense that they are associated with a group of Processes executing in a Domain and once they have been granted they may be exercised without any further checks. A limited form of revocation is possible by removing the ObjectProxy from service but this would normally result in the failure of the application using the ObjectProxy.

So far, the mechanism permits access to all the methods of an object or none at all. The rationale behind this behavior is that access to ObjectProxies is type-checked at compile time. An abstract class can be used to declare the appropriate application interface at compile time without requiring potentially inefficient checks at run-time.

The NameServer is a kernel object that provides the user interface to the ObjectProxy mechanism. It maintains a set of keys and the bindings of the keys to kernel objects. New kernel objects can be registered with the NameServer by other kernel objects.

The ObjectProxy mechanism is managed by a simple security policy that is associated with the NameServer. The policy responds to any request to access a kernel object by forwarding the request to the kernel object concerned, along with the class of the ObjectProxy expected and the Domain in which the request was made. The kernel object responds to the request by either not granting permission or by returning the address of an object. The object may be the kernel object itself or it may be an object to which it has delegated responsibility for user communication. In this way, the delegate may filter user method calls.

If the request is granted, the policy instructs the ObjectProxy mechanism to build an ObjectProxy, and this is returned to the NameServer. The NameServer binds the ObjectProxy to the key within its internal data structures and returns a pointer to the ObjectProxy to the application. Subsequent requests from the same Domain to access the same kernel object use the new key binding.

File System The security and protection mechanisms employed in the file system depend on the specialization of the file system model that is in use. In general, only memory-mapped files are directly available to an application. Access to FileStreams, ObjectContainers, ObjectDictionaries, MemoryObjects, and Domains are all provided through ObjectProxies.

The methods of file system objects augment the functionality of the NameServer at the application interface. The file system interface updates a Domain's ObjectProxy table by creating ObjectProxies directly, if the request does not violate the file system's security policy. For UNIX-like file systems, the security policy is based on the user and group identifications of the file's owner, the user and group identifications of the Domain that is requesting access to the file, and the set of access rights associated with the file. In current implementations, the address of an ObjectProxy for a MemoryObject cannot be passed from one Domain to another through shared memory and used because the ObjectProxy protection mechanism prevents one Domain from using another Domain's ObjectProxies. In future file system specializations, access lists will be used to replace the access right mechanism.

Persistent Object Store Since PersistentObjects can store References, they provide a very flexible mechanism to build protection and security schemes for accessing other PersistentObjects.

A kernel PersistentObject may use its own policy to update a Domain's ObjectProxy table using a similar scheme to that of the file system scheme described above. That policy can be based on access lists or rights of Domains stored with particular PersistentObjects or access rights that are stored in "capability-list" like PersistentObjects associated with the Domains of a user.

PersistentObjects can also be mapped into the user virtual memory of applications. In such cases, References can be used to pass "pointers" to PersistentObjects from one Domain to another through a shared PersistentObject. In such cases, the object store security policy still controls whether a Persistent Object passed by a Reference can be accessed in the new Domain.

7 Performance Issues

Choices has grown from over 30,000 to over 75,000 lines of C++ code in the current "stable" version. Systems integration of the various *Choices* designs has allowed us to take preliminary performance measurements[15]. These were measured on a NS32332, 10MHZ processor.

The performance of the object proxy approach compares favorably with the Encore Computer Corporation's UMAX (4.2 BSD UNIX) system call. The better performance of the object proxy call is a result of exploiting knowledge of the C++ virtual function calling convention. The implementation of the object proxy call avoids saving unnecessary context during the transfer from non-privileged to privileged execution.

The performance of the object-oriented file system model has been measured for the BSD 4.3 specialization of the *Choices* file system. We calibrated our measurements by repeating the same measurements on the UMAX operating system. In general, *Choices* takes slightly more time than UMAX to open or create a file but a little less time to close a file. *Choices* takes longer because its file system is memory-mapped and has additional associated data structures. Clearly, the most important operations on open files are read and write. The performance of these operations depends on whether the data is cached in main memory or written out to disk. For cached reads or writes of 8192-byte *aligned* blocks, *Choices* is faster than UMAX because it uses memory-mapped files and virtual memory support instead of a buffer cache. For uncached read and write operations, *Choices* and UMAX perform similarly because both systems are limited by the speed of the disk. *Choices* performs the lseek positioning of the file location pointer faster than UMAX, primarily because it provides a more efficient system call mechanism.

Times were also measured to copy large files from disk-to-disk and from cache-to-cache. For disk-to-disk copies, *Choices* performs slightly faster than standard UMAX, largely because of the efficiency of the *Choices* caching mechanism. For cache-to-cache copies, *Choices* completed the copy in less than half the time required by UMAX, again because of the efficiency of the *Choices* caching mechanism. *Choices* also provides a single operation, copy, to copy an entire file. By avoiding the overhead of making many system calls (2 per block copied), *Choices* provides a substantially faster file copy mechanism.

Future effort will be devoted to a more extensive analysis of the behavior of the file system and PersistentObjects under different loads.

8 Discussion

Persistent objects are the subject of much investigation[3]. The persistent objects of *Choices* are influenced considerably by the C++ implementation and have similarities to persistence in a number of systems including E/EXODUS[13, 5], O++/ODE[2, 1], PS-algol[4], SOS[17], and Comandos[10]. Because of the diversity of different schemes, the comparison of our system with other systems is an ongoing project.

Both E and O++ are based on extended versions of C++ and include object storage managers. Their primary advantages are the smooth integration of persistence within the C++ language. Our design goal has been to provide this integration without compiler modifications or extensions. E and O++ also provide many other useful features for organizing and manipulating persistent objects; for example, they both provide functions to iterate over groups of objects. They do not, however, provide security or garbage collection.

PS-algol provides distributed persistent objects, multi-user support using transactions, and garbage collection based on reachability from one or more root objects. To use persistent objects in PS-algol, a programmer must adapt his programming style to use persistent primitives for transaction support.

SOS and Comandos integrate persistent object support with customized operating systems. SOS uses an extended C++ compiler and Comandos currently uses an enhanced C compiler. They both support distributed persistent objects, but these objects are not implemented within a general file system model.

9 Summary

In this paper, we have summarized the current design of the *Choices* persistent object implementation and outlined the research in progress. *Choices* is implemented as an object-oriented system and persistent objects appear to simplify and unify many functions of the system. Our research has demonstrated that persistent data can be accessed through an object-oriented file system model as efficiently as an existing optimized commercial file system. The object-oriented file system can be specialized to provide an object store for persistent objects. Several problems arise in building an efficient persistent object scheme in a small, 32-bit virtual address space that only uses paging. Our current PersistentObject/Reference solution does have limitations, but allows quite large numbers of objects to be active simultaneously, permits sharing, and allows efficient method calls.

References

[1] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. *ACM*, 23(1):36–45, January 1989.

[2] R. Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. In *Second International Workshop on Database Programming Languages*, Oregon Coast, June 1989.

[3] M. P. Atkinson and O. P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[4] Alfred Leonard Brown. Persistent Object Stores. Technical Report Persistent Programming Report 71, Universities of St Andrews and Glasgow, October 1989.

[5] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage Management for Objects in EXODUS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–370. Addison Wesley, Reading, Massachusetts, 1989.

[6] Gary Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the USENIX Workshop on Distributed and Multiprocessor Systems*, pages 39–58, Ft. Lauderdale, Florida, September 1989.

[7] Peter W. Madany. An Object-Oriented Approach towards A General Model of File Systems. Technical Report (to be published), University of Illinois at Urbana-Champaign, May 1990.

[8] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[9] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, pages 65–79, Denver, Colorado, October 1988.

[10] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA '89*, pages 113–122, New Orleans, Louisiana, September 1989.

[11] John Ousterhout and Fred Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[12] Hal S. Render, Robert N. Sum Jr., and Roy H. Campbell. An Object-Oriented Approach to Integrated Configuration Management and Project Management. Technical Report UIUCDCS-R-89-1553, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1989.

[13] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. Technical Report Computer Sciences 824, University of Wisconsin, Madison, February 1989.

[14] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[15] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, San Francisco, California, April 1990.

[16] Aamod Sane, Roy Campbell, and Ken MacGregor. Performance of Distributed Virtual Memory in an Object-Oriented Operating System. In *Proceedings of the ACM Symposium on Priciples of Distributed Computing*, Quebec City, Quebec, Canada, August 1990 (submitted for publication).

[17] Marc Shapiro, Phillipe Gautron, and Laurence Mossieri. Persistence and Migration for C++ Objects. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 191–204, Nottingham, UK, July 1989. Cambridge University Press.