A Class Hierarchical, Object-Oriented Approach to Virtual Memory Management*

Vincent F. Russo
Roy H. Campbell
Gary M. Johnston

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, Illinois 61801–2987 USA

Abstract

The Choices family of operating systems exploits class hierarchies and object-oriented programming to facilitate the construction of customized operating systems for shared memory and networked multiprocessors. The software is being used in the Tapestry[1] laboratory to study the performance of algorithms, mechanisms, and policies for parallel systems. This paper describes the architectural design and class hierarchy of the Choices virtual memory management system.

The software and hardware mechanisms and policies of a virtual memory system implement a memory hierarchy that exploits the trade-off between response times and storage capacities. In Choices, the notion of a memory hierarchy is captured by abstract classes. Concrete subclasses of those abstractions implement a virtual address space, segmentation, paging, physical memory management, secondary storage, and remote (that is, networked) storage. Captured in the notion of a memory hierarchy are classes that represent memory objects. These classes provide a storage mechanism that contains encapsulated data and have methods to read or write the memory object. Each of these classes provide specializations to represent the memory hierarchy. They may be cached in physical memory. This paper describes the motivation for an object-oriented, class-hierarchical approach to virtual memory system design, and describes the overall architecture of such an approach, as it has been applied to Choices.

Keywords: Object-oriented design, class hierarchies, virtual memory, multiprocessors, operating systems, object-oriented operating systems, customizable operating systems, extensible operating systems.

---

*This work was supported in part by NASA under grant number NSG1471 and NAG 1-163 and by AT&T METRONET.

[1]Tapestry is funded by a NSF CISE grant.

1 Introduction

The *Choices* [2, 3] operating system architecture is motivated by the difficulties of building operating systems for specialized high-performance applications on large collections of heterogeneous shared memory and networked multiprocessors. The conventional operating system provides applications with a "kernel" that offers a predefined selection of system services that cannot be easily extended to provide specialized services for particular concurrent applications on particular parallel hardware. Choices uses object-oriented programming and class hierarchies to organize and facilitate solutions to this problem. An operating system implemented with the Choices architecture currently runs on the Encore Multimax,[2] and is currently being ported to the Intel iPSC/2[3] hypercube.

Before going into the details of the Choices memory management classes, a brief overview of the general Choices class hierarchy is appropriate. Some of the major classes in the first level of the Choices class hierarchy are shown in Table 1. Each subclass redefines and/or adds methods defined for class *Object*. Class *MemoryRange* provides the base for storage management in a Choices operating system. Instances of class *Process* are the basic units of execution in a Choices system. A Process is represented by the information necessary to execute it which includes a copy of the processor state (i.e., CPU registers) and a description of the virtual memory in which it expects to execute. Processes are scheduled and executed within a Choices system by being added to and removed from *ProcessContainers*. Class ProcessContainer is specialized to provide for Process execution and scheduling. A Process is moved from one *ProcessContainer* to another by add and remove operations in order to achieve scheduling and Process execution. Subclasses of ProcessContainer provide scheduling disciplines. Class *Exception* provides the basis for exception

[2]Multimax is a trademark of Encore Computer Corporation.
[3]iPSC is a trademark of Intel Corporation

Choices Base Classes					
Class	Methods				
Object	**ctor**	**dtor**	–	–	–
↑MemoryRange	*ctor*	*dtor*	**reserve**	**release**	**physicalAddress**
↑Process	*ctor*	*dtor*	–	–	–
↑ProcessContainer	*ctor*	*dtor*	**add**	**remove**	–
↑Exception	*ctor*	*dtor*	**raise**	–	–

Table 1: Choices Base Classes

Legend	
Symbol	Meaning
method	Definition of method.
method	Redefinition of method.
↓	Subclass or inherited method.
–	Undefined method.

handling, including traps and interrupts. The raising of an Exception causes Exception-specific movement of Processes between ProcessContainers.

In this paper, we discuss the classes within Choices that support virtual memory management. The Choices design exploits virtual memory techniques for efficient interprocess communication via shared memory. Any communication required between the applications is supported by operations on shared objects or directly via shared virtual memory. Choices support for networked multiprocessors extends the virtual memory across the network.[4] After discussing related research, we introduce the virtual memory approach adopted in Choices. Next we discuss a class hierarchy that implements this approach and outline the major methods of each of the classes. Finally, we review our design with respect to common computer architectures and summarize the status of our research.

[4]Message-oriented kernels like the V Kernel [4], Accent [9], Amoeba [11], and MICROS [12] build specific communication schemes into the lowest levels of the kernel. For example, some systems implement a few ways of providing "virtual" messages like "fetch on access." However, these systems are not easy to adapt to support other approaches such as "send process on read" or "remote procedure call on execute."

2 Problem

Virtual memory systems are used in many popular operating systems. They have met with much success at providing large address spaces on systems with limited physical memory, thereby simplifying the programming of applications with large memory requirements. Virtual memory also provides protection for a program's data and code, both within and between applications, and facilitates code and data sharing. Virtual memory systems include three major components: an address translation mechanism, a virtual memory placement and replacement algorithm, and a protection mechanism.

Each executing Process has an appropriate translation table that maps valid virtual memory addresses to physical memory addresses. Virtual memory addresses are translated by hardware into physical memory addresses. The hardware usually uses an associative cache that is loaded from the translation table to reduce the number of table look-ups it needs to perform. For physical memory allocation purposes, the virtual memory space of a process is partitioned into pages, if the partitions are all the same size (as is usually the case), or segments if not [1, 7]. Variable size segments do not form a contiguous address space. Two-level paging or segmented paging schemes may be employed in which contiguous pages are grouped into larger sections. Such schemes permit the virtual address space to be divided into non-contiguous virtual address subspaces [5, 6].

When the contents of a potentially valid virtual address is not resident in physical memory, it is stored on a backing store (such as a disk) and the translation table page or segment entry for that address is marked non-resident. A process attempting to access that address suffers a page or segment fault. The virtual memory placement algorithm retrieves the contents of the page or segment referred to by that address from a backing store and updates the translation table. Then

the instruction that caused the fault is restarted (or continued).

By itself, the memory placement algorithm may eventually fill physical memory. The memory replacement algorithm copies pages or segments back onto the backing store. Such an algorithm seeks to replace pages or segments that are not needed in the immediate future in order to minimize i/o traffic between main memory and the backing store.

The virtual memory mechanism is used in time-sharing systems to protect data from "incorrect" access by processes executing in different virtual address spaces. The mechanism may also be used to share data or read-only code between processes.

Our objectives in building the virtual memory management system for Choices are to create an object-oriented model for the entities in a shared memory multiprocessor virtual memory system and to create a class hierarchy that organizes different variants of this model for different machines and applications. In rewriting virtual memory management in a class hierarchical, object-oriented manner, we seek to provide several mechanisms for a multiprocessor environment:

- Efficient sharing of memory objects between processes executing in parallel;
- Efficient context switching between interrupt processing and user process execution;
- Very large virtual memory spaces that are bigger than physical memory;
- The use of arbitrary, multiple backing stores;
- Access to memory-mapped persistent objects whose lifetimes exceed the lifetime of an individual process or its virtual memory;
- Efficient process creation and message passing primitives that only copy shared memory

objects (such as data or code) when necessary;[5]

- The design and implementation of appropriate page replacement algorithms for memory objects;[6]

- Uniform and consistent memory management and buffering schemes that can be applied to virtual memory, such as input/output buffering, files and file buffer caches, and message caches.

Choices has adopted some of the design ideas employed in the Mach [8] virtual memory management system. In particular, Choices adopts the idea of a "memory object" (a Choices *MemoryObject*) that is cached in physical memory. Choices departs from Mach in its object-oriented, class hierarchical approach, allowing greater flexibility and customizability for given environments and applications.

Although many of the applications of the Choices virtual memory management scheme have yet to be explored, we believe the class hierarchical object-oriented approach we have adopted will allow us to attack them in a rigorous sequence of experiments.

3 Virtual Memory Framework

We will refer to the collection of data representing an addressable entity as a *memory object*. Memory objects may be defined by system and application software in Choices and include program text, stack space, disks, heaps, kernel data spaces, and files. The memory object usually resides in a

[5]That is, they employ copy-on-write shared-memory techniques to minimize unnecessary copying.

[6]Most page replacement schemes in virtual memory management systems are global. Instead, in Choices, we allow localised page replacement schemes where each memory object may have its own algorithm that optimizes the page traffic for that type of memory object.

semi-permanent form on a backing store or disk. It may also be distributed among loosely-coupled nodes connected by a networks.

Memory objects are either mapped into the virtual address space of a process or made accessable through a read/write interface (similar to the traditional notion of a file.) A virtual address mapped memory object is accessed by the processor's read/write instructions through the virtual address translation mechanism. The data of the memory object that is stored in physical memory by the virtual memory system is a cache of the object. Unlike standard virtual memory implementations, the non-resident data of each memory object is stored on its own backing store on secondary storage. When the cache releases or reuses the physical memory storing memory object data that has been modified, the backing store is updated with the data to ensure the consistency of the memory object.

In Choices, the *domain* abstraction maintains a virtual address space for a process. It assigns an virtual address range for each of the virtual memory mapped memory objects. Figure 3 shows how the addresses of a collection of virtual memory mapped memory objects form an entire virtual address space.

Each virtual memory mapped memory object has a physical memory cache of some or all of its data. Unlike conventional virtual memory systems, Choices supports a multi-level cacheing scheme for its virtual memory mapped memory objects. The domain encapsulates and hides the details of maintaining multi-level caches for each memory object and, using status information from the caches, supports a multi-level cacheing policy.

Processes may share virtual memory mapped memory objects. The domain of each process may provide its process with a different set of access rights to the same shared memory object. In a shared memory architecture, the processes sharing a memory object may share its physical cache

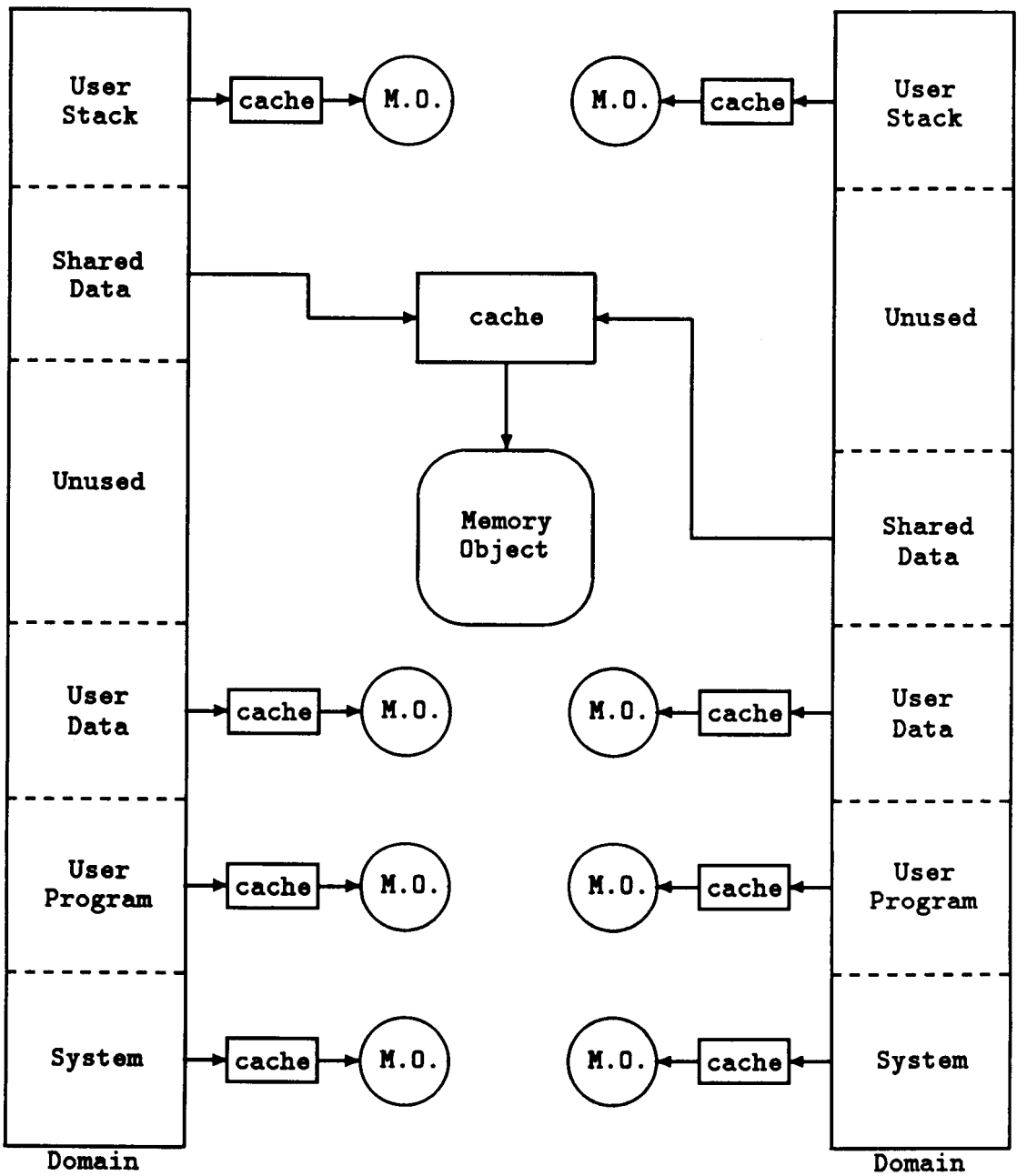

Figure 1: Conceptual View of Domains, Caches and Memory Objects

as shown in the Figure 3. Sharing across a network in a distributed system can be accomplished by having local caches on each node cache the same memory object and employing a cache consistency protocol between local caches and the memory object.

Processes may change the memory objects accessible from a domain as part of a protection scheme that implements persistent objects. (Persistent objects can exist for longer periods of time than the processes that access them.) On entry to a method of a persistent object, the domain of a process is modified to reflect the memory objects that are encapsulated by that persistent object. On exit from the method, the domain is modified to remove those memory objects that are encapsulated by the persistent object.

In the subsequent sections, we explain the concepts of memory object, memory object cache, and domain in more detail.

4 Virtual Memory Class Hierarchy

The Choices virtual memory system implementation models the components of a virtual memory management system as instances of memory management classes (objects) and operations on them (methods). These classes are written in an object-oriented programming language (C++ [10]). The components of the model are organized as a class hierarchy. In general, similar components will, if they are intended to function alike, be subclasses of a more abstract class that describes the common attributes the components inherit. This method of software organization is very powerful and supports code reuse. Some of the behavior of components in the system can be inferred by the position of their class definition in the class hierarchy. The hierarchy also allows specialization of algorithms and data structures for specific hardware needs without compromising the overall

Choices AddressTranslation Classes					
Class	**Methods**				
Object	ctor	dtor	–	–	–
↑AddressTranslation	*ctor*	*dtor*	addMapping	removeMapping	changeProtection
↑↑NS32382Translation	*ctor*	*dtor*	*addMapping*	*removeMapping*	*changeProtection*
↑AddressTranslationContainer	↑	↑	add	enable	–
↑↑NS32382MMU	*ctor*	↑	*add*	*enable*	–

Table 2: Choices AddressTranslationClasses

integrity of the design.

In this section, we introduce the class hierarchy that implements the Choices virtual memory system. The hardware dynamic address translation is managed by the AddressTranslation classes. The Choices machine independent virtual memory management scheme is implemented by the Domain, MemoryObject, and MemoryObjectCache classes and their subclasses. Subclasses of MemoryObject manage memory objects themselves while the other two classes manage memory object caches in physical memory. The Domain class allocates virtual addresses to cached memory objects. It coordinates the AddressTranslation management of the physical dynamic address translation mechanism with the logical MemoryObjectCache management of physical memory. In later sections we will describe how the classes are used to implement the virtual memory in more detail.

Subclasses of *AddressTranslation* (Table 2) provide support for hardware dependent virtual memory management. They encapsulate hardware specific address translation information, such as page tables or translation lookaside buffers. AddressTranslations also represent the hardware memory protection mechanism. They are updated by Domains and MemoryObjectCaches to fix address translation faults. Subclasses of *AddressTranslationContainer* are used to associate a processor's memory management unit (MMU) with its virtual memory page table map or translation lookaside buffer (TLB). A different subclass of AddressTranslation and AddressTranslationContainer exists for every architecture to which Choices has been ported. They are the only components of the

Choices MemoryObject Classes				
Class	Methods			
Object	**ctor**	**dtor**	–	–
↑MemoryRange	*ctor*	↑	–	–
↑↑MemoryObject	*ctor*	↑	**read**	**write**
↑↑↑MemoryObjectView	*ctor*	↑	*read*	*write*
↑↑↑↑PrimitiveFSMemoryObject	*ctor*	↑	↑	↑

Table 3: Choices MemoryObject Classes

virtual memory management system that are machine dependent and need modification when it is ported to a new architecture. The goal of our design is to restrict the functionality of these classes to a minimum in order to simplify porting and allow for the possibility of simple hardware assisted implementations in future architectures.

The *Allocator* classes are used to allocate memory (whether virtual memory or physical memory). The *Store* class is a subclass of Allocator used by MemoryObjectCaches to allocate and deallocate physical memory in the virtual memory implementation.

The *MemoryRange* classes (Tables 3 and 4) support the machine independent virtual memory system. A MemoryRange defines a finite sequence of indexed storage units. All units within a MemoryRange are the same size, which must be an integer power of two. The length of the sequence is stored and can be checked against an offset or unit index. The class can convert between byte offsets into the range and its units.

The class *MemoryObject* is a subclass of MemoryRange used to define the access protocol to the data of a memory object. Subclasses of MemoryObject define different implementations of this protocol. One such subclass, *BSDInode*, provides the MemoryObject interface to the data contained in an inode on a Berkeley UNIX file system. Other subclasses, not shown in the tables, provide the MemoryObject interface to the raw disk partitions, System V UNIX Inodes, MS-DOS files and other Choices file systems.

Memory objects may be mapped into virtual memory addresses so that the read, write, and execute hardware instructions of a processor may be used to access the data directly. Examples of such memory objects are the code and data of programs. The *MemoryObjectCache* class provides a physical cache for the data of a memory object. It uses a MemoryObject to access the memory object when it needs to fetch or store data. Cache update and writethrough is provided by the operations on the MemoryObjectCache.

The *Domain* class maintains a collection of MemoryObjectCaches together with a map of those MemoryObjectCaches into virtual memory. A Domain is responsible for the assignment of virtual memory locations to the data that the MemoryObjectCache is caching. The MemoryObjectCache itself contains no information about virtual addresses. This is important because it allows a cached memory object to be shared in several different Domains or allows a MemoryObjectCache to be accessed with different protections at different virtual address locations within the same (or another) Domain. A Domain provides methods to convert from a virtual address to a MemoryObjectCache and offset pair. This information is used by the Domain to maintain AddressTranslations.

In *paged* virtual memory systems, the MemoryObjectCache is specialized into a *PagedMemoryObjectCache*. This subclass of MemoryObjectCache maintains the cache on a page basis (parameterized by the hardware page size) and operates in a machine independent manner like a conventional paged address translation scheme.

5 Machine Dependent Details

The machine dependent portion of the Choices virtual memory management system consists of classes in two hierarchies: AddressTranslation and AddressTranslationContainer. Subclasses in

these hierarchies implement the machine dependent portion of the virtual memory system for different architectures.

An instance of class AddressTranslation is a machine dependent representation of the hardware translation tables, page tables, or translation lookaside buffer. It provides a machine *independent* interface to the rest of the memory management system. A different subclass of AddressTranslation exists for each architecture Choices is ported to, new subclasses will be added as Choices is ported to new architectures. The interface provided includes methods to add a virtual to physical translation at a given protection level *addMapping*, to invalidate the mapping for a range virtual address *removeMapping*, and to change the protection of a given range of virtual addresses *changeProtection*. The protection level arguments to these methods are machine independent and are mapped by the AddressTranslation objects to whatever protection levels are provided by the hardware.

Every Domain has an associated AddressTranslation that it uses to implement its mapping of virtual address ranges to MemoryObjectCaches. When address translation errors occur, the Domain updates its AddressTranslation from machine independent information. The Domain can determine what virtual addresses are currently valid and can request physical memory location information from its MemoryObjectCaches.

Since the complete virtual to physical mappings for a domain are actually kept in the combination of the Domain and MemoryObjectCache information, AddressTranslations are caches of currently active hardware virtual to physical mappings, much like the pmap system of Mach [8]. This allows a fixed amount of physical memory to be dedicated to machine dependent address translation.

The AddressTranslationContainer classes represent physical memory management units. There is one instance of an AddressTranslationContainer subclass per processor in a multiprocessor sys-

tem. Like AddressTranslations, a different subclass exists for each architecture to which Choices is ported. The *add* method of an AddressTranslationContainer class takes an AddressTranslation as an argument and switches the hardware dynamic address translation mechanism to use the mappings of the AddressTranslation. The *enable* method turns the hardware dynamic address translation on. The method uses its AddressTranslation argument as the initial mapping. After it is enabled, the AddressTranslationContainer will accept other AddressTranslations. Before the enable method is invoked, the processor may address physical memory directly, without dynamic address translation taking place. The enable method is used at boot time once the initial AddressTranslations are constructed.

6 Machine Independent Details

The Choices virtual memory management and page replacement algorithms are written as machine independent modules that manage memory objects and their caches. In this section, we describe the methods of the classes implementing this component of the system.

6.1 MemoryRanges

The MemoryRange class (Tables 3 and 4) defines a finite sequence of indexed storage units. Methods are provided to return the size of the units (*unitSize* and *log2UnitSize*) and the number of units (*numberOfUnits*). Methods also convert a byte offset into a unit number (*offsetToUnit*) and vice a versa (*unitToOffset*).

6.2 MemoryObjects

The class MemoryObject is a subclass of MemoryRange used to define the access protocol for data of a memory object. The primary access protocols are *read* and *write* and take arguments that specify the offset of the data and how many units are to be accessed. Subclasses of MemoryObject define different methods that implement this protocol. Current MemoryObject classes exist to represent Berkeley UNIX inodes, System V UNIX inodes, and MS-DOS files as well as other experimental file system structures currently being developed.

A memory object may be too large to reside in virtual memory (for example, a large disk). The MemoryObjectView subclass of MemoryObject provides a window into another MemoryObject. The window may be offset from the start of the MemoryObject. It uses the length inherited from MemoryRange to restrict access to the MemoryObject under the window. Methods for a MemoryObjectView allow the window to be moved. Several MemoryObjectViews may exist for the same MemoryObject.

6.3 Domains

The memory objects that the instructions of a process can access are represented by its Domain. The Domain contains a list of MemoryObjectCaches and maintains the correspondence between the virtual memory addresses and the physical memory locations used for the caching of memory object data. The Domain provides a realization of the Choices virtual memory scheme by using this information to update the hardware dynamic address translation mechanism through method calls to an AddressTranslation object.

A Domain converts a virtual address to a memory object and offset pair. The memory object is represented in the Choices system by an instance of a MemoryObject class. A MemoryObject is

Choices MemoyObjectCache and Domain Classes							
Class	Methods						
Object	ctor	dtor	–	–	–	–	–
↑MemoryRange	*ctor*	↑	–	–	–	–	–
↑↑MemoryObjectCache	*ctor*	↑	isManaged	fill	physAddr	fixFault	setProt
↑↑↑PagedMemoryObjectCache	*ctor*	*dtor*	*isManaged*	*fill*	*physAddr*	*fixFault*	*setProt*
↑MemoryObjectCacheMap	*ctor*	*dtor*	add	remove	lookup	–	–
↑↑Domain	*ctor*	*dtor*	*add*	*remove*	*lookup*	xlation	fixFault
↑Allocator	*ctor*	–	allocate	free	–	–	–

Table 4: Choices MemoryObjectCache and Domain Classes

cached by a MemoryObjectCache. The offset is used by the methods of the MemoryObjectCache to determine the physical location of the data if it is in the cache or to fetch the data from the MemoryObject and place it in the cache if it is not. A Domain can also convert a MemoryObject and offset pair back into a virtual address.

The Domain supports MemoryObjectCache overlays or multi-level cacheing. Each MemoryObjectCache overlay is kept in an ordered list of MemoryObjectCaches. When the conversion of a virtual address to a MemoryObject and offset pair is performed, the overlay MemoryObjectCaches in the list are checked, one at a time, until a MemoryObjectCache is found that can provide the offset to physical cache mapping. Using the overlays, a MemoryObjectCache and, therefore it's MemoryObject, can be copied on write from one Domain to another. A copy-on-write MemoryObject is implemented by appending a write cache after the read cache of the copied MemoryObject. The write cache will have its own MemoryObject for storing modified data.

The Domain contains a set of desired access rights for each range of a MemoryObject cached by a MemoryObjectCache. These are used to maintain the access rights for the corresponding virtual memory addresses. Some of the Domain classes are shown in Table 4.

The *add* method of a Domain binds a virtual address range to a MemoryObject offset pair. There are two forms of the add method. One binds a given virtual memory address range argument to the MemoryObjectCache. The other selects a range of virtual address that is large enough

to contain the memory object and binds the starting virtual address to the MemoryObjectCache. If necessary a MemoryObjectCache list is created and associated with the virtual address range. When a MemoryObjectCache is added at a virtual address range that already has a MemoryObject-Cache, the new MemoryObjectCache overlays the mappings of the previous MemoryObjectCache. The new MemoryObjectCache is inserted at the head of the MemoryObjectCache list. The AddressTranslation method removeMapping may be invoked to remove any outstanding hardware physical address translations that would interfere with the new MemoryObjectCache.

In our implementation, it is important that MemoryObjectCaches do not contain knowledge of the actual virtual addresses that are assigned for their caches. This permits a MemoryObject-Cache to have a cache that is mapped into different virtual memory locations in different Domains. Obviously, this will only work for a memory object that contains position independent data. Files and disks are good examples of such memory objects. The add method on a Domain also invokes a mappingList method on the MemoryObjectCache to inform it that it is has been added to a Domain.

The method *remove* deletes a MemoryObjectCache from the MemoryObjectCache list associated with a virtual address range. It deletes the mapped virtual address range and MemoryObjectCache list if the list is empty. The AddressTranslation method removeMapping is invoked to modify the hardware physical address translation mechanism for the range of virtual addresses that has been removed.

The Domain manages demand fetching of memory from memory objects and address translation errors or aborts. Upon an address translation error, the *fixFault* method is invoked with the virtual address and the type of operation (read, write, or execute) that caused the fault as arguments. The Domain translates the virtual address into a MemoryObject offset pair. It searches the Memory-

ObjectCache list for a MemoryObjectCache that can provide both a physical address mapping for the given virtual address and the appropriate access rights to the data for the requested type of operation. A MemoryObjectCache may fetch data from the memory object by invoking methods on it's associated MemoryObject instance if necessary. If a physical address is found, the Domain informs its AddressTranslation of the appropriate mapping. Otherwise, the Domain returns an error.

6.4 MemoryObjectCaches

The class MemoryObjectCache (Table 4) and its subclasses define objects in Choices that are responsible for mapping the data of a memory object into the physical memory of a computer. All, part or none of a memory object's data can be cached into physical memory by a MemoryObjectCache. The cache is filled with data by invoking methods on the MemoryObject that manages the storage of the memory object.

The *fixFault* method of a MemoryObjectCache requests a physical memory location of unit size from the Store physical memory allocator and loads the cache location from the memory object it is cacheing. (A PagedMemoryObjectCache, for example, will read a whole page of data into the cache.) It returns the physical cache location.[7]

MemoryObjectCaches may, on occasion, be forced to discard the data in a physical cache location. For example, this may occur when a Store cannot find enough physical memory to allocate to a MemoryObjectCache so that it may fix a fault. One of the active MemoryObjectCaches must release physical memory.

Each MemoryObjectCache maintains a list of all the Domains that are sharing its cache. This

[7]The *physAddr* method converts a memory object offset into the address of the physical cache location containing the data that is associated with the offset.

list is used by the MemoryObjectCache to invoke a method on each Domain to indicate that an offset within the MemoryObjectCache is no longer resident in physical memory. Information in the Domain is used to decide whether to update its AddressTranslation or not.

Each MemoryObjectCache has a maximum working set size of physical memory that it may commit to cacheing the memory object. In the present implementation, the working set sizes are determined statically and empirically. Adaptive working set algorithms are being investigated. Each MemoryObjectCache also keeps track of the current resident set of its data at any time. If new data is accessed within the memory object and there is no physical memory left in the system, one of two things will happen. If the MemoryObjectCache's current resident set size is greater than its working set size, it will free up less frequently accessed memory in the cache and re-use the physical memory to cache the newly accessed data. If the resident set size is smaller than the working set size, the MemoryObjectCache system acquires more physical memory by informing infrequently accessed MemoryObjectCaches to swap out some or all of their data.

A MemoryObjectCache may or may not manage a particular offset within a memory object. The methods *isManaged* and *manage* exists to inquire about an offset. A copy-on-write effect can be achieved using the MemoryObjectCache list and the isManaged and manage methods of the MemoryObjectCaches in the list. The original MemoryObjectCache is made read-only. A MemoryObjectCache is appended to the list that initially manages none of the range it represents and has an empty MemoryObject for backing store. The Domain invokes the changeProtection method on its AddressTranslation to make all the range of the MemoryObject *read only* in hardware. When a write fault occurs, the data in the unit of the read-only MemoryObjectCache are copied into the appended MemoryObjectCache which then manages future accesses to the unit. The appended MemoryObjectCache returns the new physical address. The Domain uses this physical

address to update the AddressTranslation.

7 Conclusions

This paper described the design and implementation of the virtual memory system for the Choices operating system. The implementation is based on an object-oriented architecture and is implemented in an object-oriented language. The software is organized by a class hierarchy.

The object-oriented approach has little impact on performance. The software is coded in C++ which provides a very efficient implementation of method invocation and inheritance. The virtual memory scheme benefits from an object-oriented approach because it allowed us to prototype different implementation schemes, reuse code, maintain machine independent abstractions, encapsulate implementation decisions, separate policy from mechanism and provide for common interface.

In Choices, the virtual memory concepts are derived from many existing systems. However, their implementation in an object-oriented architecture is, we believe, new. The use of object-oriented design and programming to build an operating system is also significant.

The design of our system permits paging and page replacement to be tailored to a MemoryObject and its cache. This should permit us to experiment with cache replacement and data fetching algorithms that can take advantage of access patterns that are particular to a type of memory object. The Domain manages the implementation of the machine independent virtual memory management schemes using machine dependent dynamic address translation mechanisms. MemoryObjectCaches are independent of the virtual addresses that are used to access data in the caches and this allows them to be easily shared or even relocated. For efficiency and additional functionality, the Domain supports a scheme of overlayed caches. This scheme supports copy on write, recovery caches for

fault-tolerant computing and local and global memory hierarchies.

Choices is currently being ported from the Multimax to an Intel iPSC/2 hypercube. The memory management system is being adapted to use shared networked virtual memory. When this is complete, a further account of the Choices virtual memory system will be written.

References

[1] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[2] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[3] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[4] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[5] Intel Corporation, Santa Clara, California. *80386 System Software Writer's Guide*, 1987.

[6] National Semiconductor Corporation, Santa Clara, California. *Series 32000 Databook*, 1986.

[7] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1985.

[8] Richard Rashid et al. Machine–independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

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

[10] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[11] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *ACM Operating Systems Review*, 15(3):51–64, July 1981.

[12] L. D. Wittie and A. Van Tilborg. MICROS – a distributed operating system for MICRONET – a reconfigurable network computer. In H. A. Freeman and K. J. Thurber, editors, *Tutorial: Microcomputer Networks*, pages 138–147. IEEE Press, 1981.