The Spring Virtual Memory System

Yousef A. Khalidi
Michael N. Nelson

SMLI TR-93-9 February 1993

Abstract:

In this document we describe the architecture and the implementation of the Spring virtual memory system. The architecture separates the tasks of maintaining memory mappings and protections from the task of paging memory in and out of backing store. A per-node virtual memory manager is responsible for maintaining the mappings on the local machine while external pagers are responsible for managing backing store. A novel aspect of the architecture is the separation of the memory abstraction from the interface that provides the paging operations. The design supports flexible memory sharing, sparse address spaces, and copy-on-write mechanisms. Support for distributed shared memory and extensible stackable file systems are natural consequences of the design. The architecture is implemented and has been in use for over two years as part of an experimental operating system.

Categories and Subject Descriptors:

D.4 Software:
[**Operating Systems**]: D.4.2 Storage management
D.4.3 File systems management
D.4.6 Security and protection
D.4.7 Distributed systems

Additional Keywords and Phrases: Object-oriented operating system, Distributed shared memory, Memory coherency, Micro-kernel

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
yak@eng.sun.com
mnn@eng.sun.com

The Spring Virtual Memory System

Yousef A. Khalidi, Michael N. Nelson

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Spring is an experimental object-oriented operating system developed by our research group at Sun Microsystems Laboratories. In Spring the object paradigm pervades and unifies the system. The system is structured around a small nucleus that provides the basic mechanisms for object invocation and thread control. Traditional operating system functionality (such as file system services) is built on top of the substrate provided by the nucleus as user-level applications. Entities in the system are represented by objects, and services are requested by invoking objects. Spring is a distributed multi-threaded system that is constructed to exploit a range of systems from tightly-coupled multiprocessors to more loosely-coupled networks. Spring supports traditional UNIX® programs through compatibility mechanisms [1], but it is aimed toward new computing requirements, such as transparent distribution, high reliability, and stronger security.

The architecture and implementation of the virtual memory system of Spring is presented in this document. The design follows the basic Spring object model and strives to meet the requirements of the diverse intended applications of Spring. An accompanying document [2] describes the implementation of the Spring file system, an important client of the virtual memory system.

The Spring virtual memory system provides:

- Flexible, distributed, and secure memory mapping and sharing.
- Well-defined object-oriented interfaces for external (user) pagers.
- Support for distributed shared memory.
- Support for stackable file systems.
- Efficient bulk-data transfer mechanisms.

The rest of this section provides an overview of Spring. The goals of the virtual memory system design are listed in section 2. Related work is described in section 3. Section 4 gives an overview of the virtual memory system and lists its basic components. The virtual memory manager is described in section 5. Section 6 discusses the objects implemented by the external pagers. Section 7 details an important protocol used between the virtual memory system and the pagers. Section 8 gives a brief overview of the Spring file system, an important client of the virtual memory system. Implementation details of the virtual memory system are presented in section 9. Finally, conclusions and planned extensions to the architecture are listed in section 10.

1.1 The Spring Operating System

Spring is a distributed, multi-threaded operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of methods to manipulate that state. The description of the object and its methods is an *interface* that is specified

in an *interface definition language*. The interface is a strongly-typed contract between the implementor (*server*) and the *client* of the object.

Spring strives to keep a clear separation between *interfaces* and *implementations*, and in general there is no special status for interfaces that are provided as part of the base system. The Spring system is perceived as a set of interfaces rather than a set of implementations.

A Spring domain is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the clients of other objects. The implementor and the client can be in the same domain or in a different domain. In the latter case, the representation of the object includes an unforgeable nucleus *handle* that identifies the server domain.

Since Spring is object-oriented it supports the notion of *interface* inheritance. An interface that accepts an object of type *foo* will also accept a subclass of foo. For example, the *address space* object has a method that takes a *memory* object and maps it in the address space. The same method will also accept *file* and *frame_buffer* objects as long as they inherit from the memory object interface.

The Spring kernel supports basic cross domain invocations, threads, and low-level machine-dependent interrupt handling, and provides basic virtual memory support for memory mapping and physical memory management. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a network proxy server. In addition, the virtual memory system depends on external pagers to handle storage and network coherency.

A typical Spring node runs several servers in addition to the kernel (Figure 1). These include the domain and virtual memory managers; a name server; a file server; a linker domain that is responsible for managing and caching dynamically linked libraries; a network proxy that handles remote invocations; and a tty server that provides basic terminal handling as well as frame-buffer and mouse support. Support for running UNIX binaries is also provided [1] .

The Spring file system supports cache coherent files [2]. The file object interface inherits from the memory object interface and therefore can be memory mapped. The file system uses the virtual memory system to provide data caching and uses the operators provided by virtual memory caches to keep the data coherent. The file system also acts as the system pager (see section 8).

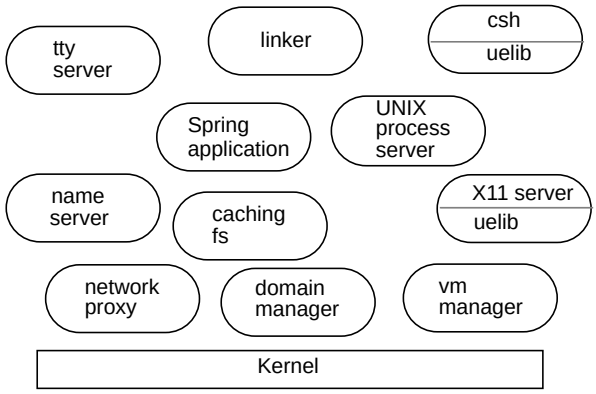

Figure 1. Major system components of a Spring node

2 Design Goals

The design of the virtual memory system is motivated by the following considerations:

- **Spring object model.** The virtual memory system should adhere to the Spring object model and should meet the general goals of Spring, such as security and extensibility.
- **Separation of concerns.** Separation of policy *vs.* mechanisms is a very important concept in Spring that the virtual memory system must follow. The virtual memory system should provide a basic set of mechanisms but should as much as possible leave the policies of using these mechanisms to other entities in the system.
- **Distribution.** In Spring, object invocation is location transparent and distribution is the default. The virtual memory system facilities must be usable regardless of their location. Any two address space objects (see section 4) may share a range of memory between them. Since the address spaces may be on different nodes, distributed shared memory must be supported.

There are three important consequences of our design goals:

- Any client should be able to implement the abstraction of memory objects. The client could be located on a local or remote machine.
- The base system must not put arbitrary trust in the implementor of a memory object. The implementor of a memory object is allowed to handle its own memory only.
- Since memory objects are abstractions of memory, the architecture should allow for different views on the same memory to be implemented efficiently. In an object-oriented system, it is desirable for a memory object to encapsulate other properties, such as protection properties (e.g., read-only *vs.* read-write). An implementor of a memory object may want to hand out a read-only memory object to one client while giving another client a read-write memory object, where both objects encapsulate the same underlying memory (e.g., see [2]). When both clients map their respective memory objects on the same machine, each should be able to do so given the protection restrictions embodied in the memory objects. Moreover, mapping the memory objects should cause the system to use the same underlying memory cached at the node.

3 Related Work

There are several systems that provide rich virtual memory subsystems that support the notion of external pagers and distributed shared memory [3][5][6][7][8][9]. An example of an early system that provides flexible memory mapping support is Multics [10]. Apollo Domain [11] is an early commercial system that supported the notion of distributed virtual memory. The Choices [12] system uses an object-oriented framework for building the operating system but it has a more traditional virtual memory system with no support for distribution or external pagers. In this section we concentrate on comparing our virtual memory system to MACH and to traditional commercial UNIX as represented by SunOS/SVR4.

3.1 MACH

The MACH operating system [3][4] has a flexible virtual memory system that supports an external pager interface. Our system differs from MACH in several aspects:

- Spring provides different views on the same memory. In our system we have the ability to encapsulate different access rights in different memory objects that represent the same underlying memory. The memory objects can then be handed out to different clients. Clients can then map their respective memory objects on the same node such that:

  each mapping cannot exceed the protection restrictions embodied in the memory objects (e.g., a read-only memory object cannot be mapped read-write),

  each mapping uses the *same* physical memory cached at the node,

  and each mapping is done in a manner that does not involve a third trusted agent between the virtual memory system and the implementor of the memory object.

  To achieve a similar effect in MACH one has to:

  a. have a copy of the data per memory object and force the pager to copy the data between the different memory objects even on the same machine, or

  b. develop a protocol based on a third trusted agent that sits between the system and the client (e.g., see [4], page 103), or

  c. modify the external pager interface, perhaps along the lines of our system.
- Spring separates the memory object from the object used for paging operations (the pager object). In MACH these two objects are one and the same although they provide different functionality: the first encapsulates access to a (logical) piece of memory while the other is used to obtain the physical underlying memory. Such separation gives pagers the flexibility to implement the memory object and the pager object in different domains. Our file system benefits greatly from this ability as described in [2] (see also section 4.4).
- Spring provides an architecture tailored for composing (or stacking) file systems.
- Spring uses strongly-typed object-oriented interfaces. As with all other Spring interfaces, the virtual memory system interfaces are specified in a strongly-typed object-oriented definition language that provides interface inheritance. The base MACH system deals with virtual memory objects at the lower level of ports with object-*based* interface language support but with no support for object-*oriented* interface inheritance.

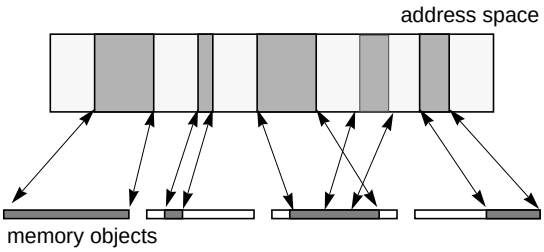

Figure 2. User' s view of address spaces.

An address space is a linear range of addresses with regions mapped to memory objects. Each region is mapped to a (part of) a memory object. Each page within a mapped region may be mapped with read/write/execute permissions and may be locked in memory.

- Spring provides *move* semantics for bulk data transfer during object invocation in addition to *copy* semantics (see section 9).

3.2 SunOS/SVR4

SunOS/SVR4 is representative of current state of the art commercial UNIX virtual memory systems [13]. It supports memory-mapped files between different processes on the same node, private mappings (which are similar to traditional copy-on-write), and access protection selectable on a per-page basis. The system, however, does not provide an external pager interface (clients cannot create new memory objects), coherent file mapping across the network, a general stackable file system architecture, nor efficient bulk data transfer between processes.

4 Overview of the Spring VM System

In this section we present an overview of the virtual system architecture.

4.1 Basic objects: memory and address space objects

Most clients of the Spring virtual memory system deal with only two types of objects: *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain while a memory object is an abstraction of store (memory) that can be mapped into address spaces.

4.1.1 Memory Object

An example of a memory object is a read-only file object obtained from some file system. The same memory object can be mapped into more than one address space at the same time on more than one machine. Note that the memory object interface does *not* provide page-in and page-out methods: a holder of a memory object can either map it into an address space or pass it to another client. The significance of not providing paging operations on the memory object is explained in section 4.4.

4.1.2 Address Space Object

Each Spring domain has an address space object that represents its virtual address space. As with other Spring objects, an address space can be operated on by any client that holds the object. Appropriately authenticated clients can obtain and operate on the address space object of other domains, regardless of their location.

The address space has ranges of addresses that are mapped to (parts of) memory objects, called *regions*. Each page in a region may be mapped with read/write/execute permissions and may be locked in memory. The address space can be sparse. Figure 2 shows the user's view of an address space. Appendix A lists the methods of the address space object.

The main operations on address space objects are to map and unmap (part of) memory objects into selected address ranges of the address space. Since a memory object encapsulates a maximum access mode, a client may map a memory object as long as the requested access mode does not exceed the maximum access mode of the object.

Operations are also provided to allocate new *zero-filled* memory. Zero-filled memory is generally used for stack and heap regions, and is backed by anonymous memory objects.[1] Zero-filled memory is normally allocated from a per-address space default anonymous memory object. Zero-filled memory can also be allocated using an address space operation that obtains-and-maps anonymous memory objects.

1. Anonymous memory objects are either obtained from the system swap pagers or optionally from a per-address space swap pager (see section 6.3).

Address space objects provide additional functionality including:

- An operation to make a copy of a memory object and to map the copy into an address space. The copy operation is implemented as copy-on-write.
- The ability to lock in physical memory specific virtual address ranges of the address space. For this operation to succeed, an appropriately authenticated address space object must be used.
- Other housekeeping functions, including the ability to catch and notify a handler object of translation-faults due to access violation and accesses to unmapped addresses. The handler object can also be notified of changes to address space mappings. Also, there are methods to get a description of mapped regions, and to query memory usage statistics. These functions are useful for garbage collection and debuggers.

4.2 Major players

There are two types of servers that co-operate to provide the implementation of address spaces and memory objects: a per-node *virtual memory manager* (VMM) and *external pagers*. The VMM is the implementor (type manager) of address space objects, while memory objects are implemented by pagers. A pager supplies (pages in) and stores back (pages out) the actual contents of the memory objects. Any client program can act as a pager.

A VMM presented with a request to map a memory object into an address space has to be able to obtain the actual memory represented by the memory object, since the memory object itself does not provide methods for obtaining this memory. Therefore, the VMM contacts the pager that implements the memory object by invoking the *bind* method on the memory object. The purpose of the bind operation is to point the VMM to a local data cache that provides the contents of the memory object.

4.3 Cache and pager objects

The VMM and the pager exchange two objects during the bind operation: the *pager* object and the *cache* object. The pager object provides methods to page in and out memory blocks and is used by the VMM to populate a local memory cache. The cache object is implemented by the VMM and is used by the pager to affect the state of the cache.[2]

A given pager object—cache object pair constitutes a two-way communication channel between a pager and a virtual memory manager. Typically, there are many such channels between a given pager and a VMM.

4.4 Separating the memory object from the pager object

Unlike more traditional systems such as MACH, the Spring memory object does not provide paging methods. Table 1 summaries the differences between a MACH memory object and a Spring memory object.

.In Spring the memory object *represents* memory and is separated from the pager object that actually *provides* the methods to page-in and page-out the memory.

This separation has the major advantage of giving the implementor of the memory object the power to place the implementation of the memory object in a separate domain from the implementor of the pager object.

For example, the Spring file system uses this separation to interpose a local attribute caching file system (CFS) in the local node, with the end result that all file attributes are cached by the CFS, file data is cached by the VMM, all reads and writes to the file go to the local CFS and use the data cached by the VMM, yet all page-ins and page-outs go directly to the remote server where the data is stored on disk [2]. The file system also ensures that all bind opera-

TABLE 1. Memory object in MACH and Spring

	MACH	**Spring**
Memory Object	• memory mapped • init/terminate ops • paging operations	• memory mapped • bind operation • *no* paging operations
File Object	• same port as memory object • may provide file read/write operations	• inherits from memory object • provides file read/write operation • *no* paging ops

2. The cache object is actually part of the *cache manager* interface. In this paper, we concentrate on one particular implementation of the cache manager interface, the one implemented by the VMM.

tions go to the CFS, thus changing the bind into a local call instead of a remote operation. We also utilize the bind operation and the separation between memory and pager objects in our extensible file system architecture.

Note that two or more memory objects can encapsulate the same underlying memory, but with different access rights. As far as the VMM is concerned, each memory object is unique; the VMM relies on the memory object's pager to point it to a data cache from which the VMM obtains the contents of the memory object. This extra level of indirection allows different memory objects that share the same pages (but perhaps encapsulate different access rights) to have their pages cached in the VMM instead of flushing the same pages back and forth between the VMM and the pager.

4.5 Summary of VM objects

In summary, the main components of the virtual memory system are (Figure 3):

- *address space* objects are implemented by the VMM. There is one address space object per Spring domain. Operations on address space objects include mapping and unmapping memory objects into the address space.
- *memory* objects are abstractions for memory. Memory objects are implemented by pagers that are responsible for servicing page-in and page-out requests from VMMs.
- *cache* objects are containers of physical memory. They are created by virtual memory managers as a result of requests by external pagers. These objects cache memory pages that belong to pager objects. Pagers instruct the VMM to obtain the contents of memory objects from local cache objects during the bind operation.
- *pager* objects are used to build a two-way communication channel between the VMM and the pager of a cache object. Pagers implement pager objects. The VMM pages data in/out of their cache objects from/to the corresponding pager objects.

The next two sections describe the cache and pager objects in more detail by discussing the functionality of their object managers. Note that, strictly speaking, the virtual memory architecture is defined in terms of the objects listed above and not the servers; implementations are free to use more than one domain to provide these objects (see section 8).

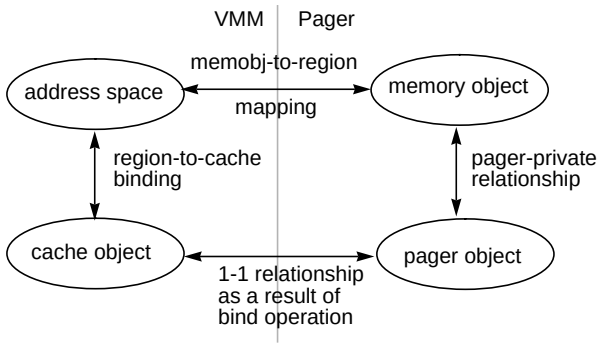

Figure 3. Relationship among basic VM objects

5 Virtual Memory Manager

There is a single virtual memory manager per Spring machine. The functions of the VMM can be grouped into the following categories:

- The VMM is the type manager for address space and cache objects. It is responsible for maintaining physical memory and paging memory in and out of pager objects.
- The VMM is responsible for handling the memory management hardware, including the memory management unit (MMU) and any hardware data caches.

The VMM also exports *vmm* objects that are used by pagers to create cache objects at the VMM. The vmm object inherits from the *cache manager* interface and provides methods for creating cache objects (Appendices B and C). Section 9 describes our current implementation of the VMM. The rest of this section describes the interface of the cache object in detail.

5.1 Cache Objects

Cache objects are implemented by the VMM. A cache object provides methods to cache memory blocks that belong to pagers. The underlying state of a memory object cached at a particular VMM is encapsulated in a cache object.

Pagers request the creation of cache objects at a VMM during the process of binding a mapping from an address space region to the underlying state of the memory object. When the VMM is presented with a memory object to map, it invokes the *bind* method on the object. This operation returns (among other things) a pointer to a locally-implemented cache object which the VMM uses to obtain

the actual pages that back the virtual memory in the mapped region. Each cache object has an associated pager object. Section 7 describes this binding protocol and the rationale behind it in detail.

Data is paged in/out of a cache from/to the corresponding pager object. The cache object provides a set of operations for pagers to control the cache. Each VMM is responsible for managing the physical memory at its local node. The VMM's view of physical memory is centered around the contents of cache objects.

A holder of a cache object (normally the pager) may destroy it, and any pages held in the cache when it is destroyed are discarded. A cache object can also be destroyed by the VMM at anytime that it is empty and it is not currently mapped. The pager, if it wishes, can be informed when the last binding to a cache object is gone.

A cache object holds data blocks. Information regarding the size of the data block is sent by the pager to the VMM at cache creation time. The pager also sends other information at that time, including hints on how the cache will be used.[3] The VMM may transmit one or more blocks per call (the interface specifies the starting offset in the cache and a length aligned to the block size).

A data block may be obtained by the VMM in one of two modes: *read-only* or *read-write*. The VMM and the pager may issue requests to each other to "upgrade" read-only blocks to read-write and "downgrade" read-write blocks to read-only.

A cache object has an ownership state that is under the control of the pager that creates the cache. There are two ownership states: *exclusive* and *shared*. The intended use of the ownership state is for the pager to set it to exclusive unless the pager knows that the contents of the cache is shared with some other cache (most probably on a different machine). The ownership state gives the pager control of whether the VMM should invoke the "usual" optimization of upgrading read-faults on read-write regions to read-write page-ins.

3. Currently, the hints include an indication of whether the cache holds text, data, or anonymous memory, and an indication of the cost of using the pager.

The ownership may be changed by the pager at any time. The state of the cache is a *hint* to the VMM that modifies the behavior of the VMM during page-in. The VMM may request the page from the pager object in mode read-write when the following is true:

- the state of the cache is exclusive,
- the VMM is in the process of satisfying a read page-fault, and
- the region where the fault occurred is mapped read-write.

If the cache state is shared, then a read page-fault results in a request for the page in mode read-only. This is the only difference in the VMM behavior between the two states.

The cache object provides several methods that are used by the pager to control the cache. Since the pager must be prepared to receive the data stored at the VMM at any point, other calls from the VMM to the pager may complete before the call requesting the data returns. The VMM guarantees that when the call returns successfully the data requested is at the pager. The requests issued by the pager to the cache object are:

- flush_back: used to remove data from the cache and send it back to the pager.
- deny_writes: used to "downgrade" read-write blocks to read-only. Any modified blocks are returned. A copy of the data is retained in the cache in read-only mode.
- write_back: used to request a copy of all modified blocks. Data is retained in the cache in the same mode as before the call.
- delete_range: used to remove data from the cache. No data is returned
- zero_fill: used to indicate that a particular range in the cache is zero-filled. The data blocks indicated by this range are held by the VMM in read-write mode.
- populate: used to introduce data blocks into the cache.

If a call from the VMM to a pager object takes "too long" or the call fails with an unexpected exception, the VMM may terminate the call and destroy the cache and all bindings to it; page-out the data through its default pager(s); or block "forever" waiting for pager to finish. The decision is made depending on the privilege of the pager[4]. For example, the VMM will block waiting for operations on the default pager(s).

6 Pagers

Pagers compliment the functions of the VMM by providing operations for fetching and storing the contents of pager objects. Whereas the virtual memory manager is concerned with maintaining the mappings between address spaces and cache objects and with reflecting these mappings in the underlying address translation hardware, pagers are concerned with providing and storing the contents of memory objects (by providing paging operations on the corresponding pager object). Appendix D lists the important methods of the pager object and memory object interfaces.

Note that we use the term “pager” to refer to the implementor of pager and memory objects. As noted before, the implementations of these objects can reside in different domains. As far as the VMM is concerned it deals with pager and memory objects and it does not care where the implementations of these objects reside.

6.1 Pager object

The VMM issues several requests to the pager object. For each request, the VMM specifies:

- the starting offset in the cache and a length (both aligned to the block size of the cache),
- whether the request is for read-only or read-write data, and
- the current cache ownership state.

Data is transmitted by using the normal Spring bulk data passing mechanism (section 9.6). As an optimization, if the data is null (all zeros), an indication is sent instead of the actual data. The requests issued by the VMM to the pager object are:

- page_in: used to request data be brought into the cache.
- page_out: used to move data out of the cache.
- write_out: used to change the state of read-write data to read-only. Data is retained in read-only mode in the cache. Only called for read-write data.
- sync: used to copy data back to the pager. Data is retained in same mode.

4. The privilege of the pager is encapsulated in the vmm object that the pager must obtain in an authenticated manner before creating any cache objects at the VMM.

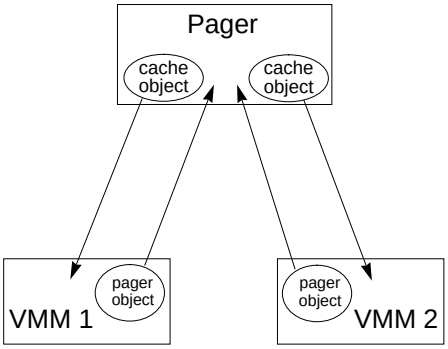

Figure 4. DSM coherency maintained by pager

For clean (i.e., not modified) blocks and for cache objects that are not bound writable (i.e., no read-write mappings exist to these caches), the VMM will never transmit the actual data in page_out, write_out or sync calls. The pager is at liberty to cease communicating with a misbehaving VMM that is violating the protocol.

Note that the system is designed such that the VMM can always reuse a clean physical page without the need to inform the pager. This is an important consideration in the design of a virtual memory system that uses external pagers. To require the VMM to inform the pager each time a clean page is reused would severely limit the page reclamation policies of the virtual memory system.

6.2 Network Virtual Memory

Network-wide virtual memory—also known as distributed shared memory (DSM)—refers to the case where the same memory is mapped on more than one machine simultaneously. A pager, if it chooses, may service requests from more than one virtual memory manager for the same memory object. Therefore, pagers are responsible for maintaining the consistency of memory objects’ data stored in different cache objects (Figure 4).

VMMs do not know about DSM; they call the pager object associated with a given cache object when a particular page is needed. The pager is the only entity that knows about distribution. In particular, the interfaces to the address space and cache objects are designed such that the VMM does not need to contact other VMMs to implement its functions. This is in line with our goal of maintaining the separation of concerns between the VMM and the pager.

Methods on the cache object are provided for a pager to request back memory pages from the VMM and to change their access mode to read-only to facilitate the implementation of multiple-reader single-writer consistency protocols. Our file system implements network coherent mapped files using such a protocol; see section 8.

The coherency protocol is not specified by the architecture. Pagers are free to implement whatever coherency protocol they wish. The cache-pager objects channel (sections 5.1 and 6.1) provides the basic building block for constructing the coherency protocol.

6.3 Swap Pagers

The VMM requires anonymous memory objects to satisfy zero-filled memory requirements, for example, as a result of the *allocate_memory* and *copy_and_map* calls on address space objects. The swap pager interface allows the VMM to request anonymous memory objects from swap pagers. A system swap pager is added by invoking the vmm object *add_swap_pager* method. Normally at boot time one or more swap pagers are added. Swap pagers can be added at any time.

Each address space has one default anonymous memory object that is normally used to satisfy zero-filled memory requirements. In addition, a domain may set its own swap pager by calling the *set_swap_pager* method on its address space object.

Note that anonymous memory objects are just like any other memory object in that they can be passed around and mapped in other address spaces. The VMM deletes the memory object when it is unmapped from the address space (client applications may of course clone the object; the VMM deletes its own copy only).

7 The Bind Protocol

When the VMM is presented with a memory object to map into an address space, it needs to identify a cache object—pager object "channel" that can be used to page-in and page-out the contents of the memory object. The purpose of the bind protocol is to create and/or identify a paging channel for each memory object.

7.1 Rationale

One design possibility is to associate a global identifier with each memory object. Each time the VMM is asked to map a memory object, it uses the global id to see if there already exists a paging channel for this memory object. If no such channel exists, it contacts the memory object to establish a channel. This is basically the approach taken by the MACH virtual memory system. The drawback of this scheme is that it does not allow two distinct memory objects that encapsulate the same data to use the same channel.

The approach we decided on is the following: When presented with a memory object, the VMM asks the memory object through the bind operation to provide a pointer to an existing cache object—pager object channel that can be used for paging purposes.[5]

After a paging channel is identified the VMM can satisfy the requests for mapped memory (i.e., faults) from the cache. The VMM can populate the cache by invoking the pager object and the pager can request back any of its data by invoking the cache object, as described before in sections 5.1 and 6.1.

7.2 Protocol description

Figure 5 shows the sequence of operations made during the bind operation. The step numbers correspond to the following list:

1. An application issues a request on an address space object that requires mapping a memory object to a region in the address space (or a request to make a copy of a memory object).
2. The VMM is presented with a memory object to map (or to make a copy from the object). It needs to associate the mapping with a local cache object. Therefore, it calls the *bind* method on the memory object requesting from the pager a pointer to a cache object to use when accessing the mapped memory. The arguments of the bind request include the name of the VMM (*not* the object representing the VMM; see below), the length, and the access mode of the requested binding.

5. For performance reasons, we insure that the bind calls are local to a given machine by using the caching file system. See section 8.

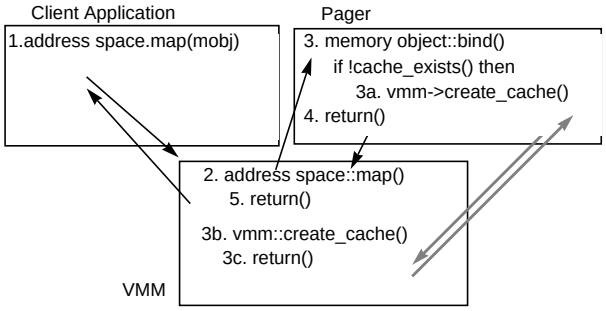

Figure 5. The bind protocol

(1) Client requests mobj to be mapped in an address space, (2) the map request is turned around into a *bind* on mobj. (3) If a cache that backs mobj does not exist at the calling VMM then (3a) a new cache is created at the VMM, etc. (4) The *bind* call returns pointing the VMM to a local cache object. (5) The VMM uses this cache object to back the requested address space region.

3. The pager that implements the memory object receives the bind request. It decides whether or not a cache object that caches the state of the memory object already exists at the requesting VMM.

3a. If no cache object that caches the contents of the memory object exists at the VMM, a *create_cache* call is issued to the VMM. [6]

3b. The VMM receives a *create_cache* call that includes a pager object as an input argument.

3c. The VMM returns a cache object plus a list of *cache-rights* objects. A cache-rights object is a Spring object that represents the right to access a cache object with an encapsulated access right. It is used as a secure capability.

4. The pager returns from the bind call by pointing the VMM to an existing local cache object that caches the contents of the memory object. The "pointer" used by the VMM is a cache-rights object and not the cache object itself.

5. The VMM uses the cache-rights object to find the corresponding cache object and completes the original client request by checking the requested access mode against the access mode encapsulated in the cache-rights object.

7.3 Cache-rights object

The cache-rights objects returned in step 3c above are used as secure capabilities. A cache-rights object encapsulates an access right to a cache object and supports one method: to obtain other cache-rights objects that encapsulate a *subset* of the encapsulated access rights. The VMM supports four possible access rights:

- read-only
- read-execute
- read-write
- read-write-execute

A cache creation request from the pager includes the maximum access rights of the cache being created. The VMM returns at least one cache-rights object that encapsulates the maximum requested access right. The holder of the cache-rights object may obtain weaker versions of the object by calling the *create_restricted_sibling* method on the object. For example, a read-only cache-rights object can be obtained from a read-write cache-rights object, but a read-write-execute object cannot be obtained from a read-write cache-rights object. The rationale behind using the cache-rights object is explained in the next section.

7.4 Discussion

We argue in this section for the correctness of the protocol:

- When a VMM is given a memory object to map, it needs to find a corresponding cache object. The only entity it can ask this question to is the memory object itself.
- However, the VMM must be sure that when it asks the memory object, "Give me a cache object that has your data," that the answer points to the correct cache object and not to some other cache object thus compromising security. The VMM does not trust pagers to be honest. The VMM trusts pagers only with the data they are supposed to manage. [7]
- Therefore, the VMM protects itself by requiring the pager to return as a result of the bind call a cache rights object and not a forgeable identifier, since a forgeable

6. The pager first looks up the *vmm* object given the name passed in the original bind call if it does not have the VMM object cached already. The name lookup is made on some well-known name server in an authenticated manner.

7. If a pager does not even handle its own data correctly, then the only losers are those clients that depend on its memory objects. Looking up a memory object from a pager in a secure manner is the responsibility of these clients and not the responsibility of the VMM.

identifier can give a malicious pager access to a cache that it should not control.[8] Similarly, the pager protects itself by returning a cache-rights object and not the actual cache object since a cache-rights object is of use only to the implementor of the cache object and to nobody else. Note that a cache-rights object is a Spring object and is not forgeable.

- When a pager receives a bind request from a VMM it does not know for sure that the caller is really the VMM indicated in the call. Moreover, pagers do not trust the VMM except for handling the data of the memory object in question.
- Therefore, the VMM sends its name and not a VMM object in the bind request. It is up to the pager to use the name to look up an authenticated VMM object. Once an authenticated VMM object is obtained, the pager can issue a *create_cache* call knowing with certainty that it is invoking the right VMM. Note that pagers can look up an authenticated VMM object once and cache it for future *create_cache* calls.

If the system is structured such that the pager, the VMM, and the original client are on the same node and a cache object already exists at the VMM, an address space *map* request can be satisfied by issuing two local object invocations: the address space *map* call and memory object *bind* call. Our file system uses such an implementation (section 8).

7.5 Cache Reclamation

Pagers may delay deleting unattached caches in the hope of reusing them later on. The virtual memory system tries to keep as many unattached caches as it can. However, as with any resource, the system has to impose a limit on the maximum number of unattached caches. Therefore, the VMM has to reclaim some of these caches when the limit is reached.

It is possible that while a pager is returning from a *bind* operation the VMM may decide to reclaim the same cache referenced in the call. Since it is not acceptable to fail the *bind* call (and the corresponding address space *map* call), the bind protocol needs to be extended to recover from this race. Although this race is seldom encountered in practice, a recovery protocol is necessary nonetheless.

The protocol extension is the following: When the *bind* call returns, the VMM checks to see if the cache-rights object points to a valid cache. If it does not, then instead of failing the *map* call, it invokes the *final_bind* method on the memory object, passing in addition to the usual bind arguments a *bind-key* object which has no methods. Note that at this point the VMM does not know whether it has hit a cache reclamation race or it is simply dealing with a malicious/incompetent pager.

When the pager receives the *final_bind* call, it is expected to call the VMM passing the bind-key object to the *create_cache_object_and_bind* or the *bind_cache* call. The pager calls the latter method when it believes that it has a cache at the VMM. If the *bind_cache* call fails, the pager then calls the *create_cache_object_and_bind* method.

When the VMM receives either call it uses the passed bind-key object to identify the outstanding *final_bind* call and associates the new cache with that call. A successful execution of a *bind_cache* or a *create_cache_object_and_bind* guarantees that a cache is bound to a bind-key object and that this cache will not be deleted until the bind-key object is deleted.

When the *final_bind* call returns to the VMM, it checks to see if a *create_cache_object_and_bind* or a bind_cache was executed successfully. If so, the original *map* request is satisfied, otherwise the VMM fails the *bind* and the corresponding *map* request.

8 Spring File System

An important client of the virtual memory system is the Spring file system [2]. The Spring file system supports *file* objects that inherit from the *memory, io*, and *authenticated* objects. Therefore, Spring files can be memory-mapped just like any other memory object, and can be operated on as an io stream. Files also encapsulate a principal name and an access control list (ACL).

The file system caches all file data in VMM cache objects (i.e. there is no double caching of the same data in the file server and the VMM). In addition, the file system makes

8. Alternatively, one could send an encrypted identifier. As with other parts of Spring, we made a conscious decision to avoid using encryption and instead used a Spring object. This way we hide encryption, if any, in the support the system provides for secure Spring objects and not in application code.

sure that all file read and write calls are coherent with memory-mapped accesses to the same file. The file system implements reads/writes to a file by mapping (part of) the file into its address space and handling the bind request itself. The file system also keeps all file attributes coherent (e.g. file length).

The file system supports network-coherent files. It implements a multi-reader/single-writer per-block coherency protocol. It acts as a pager and uses the pager-cache methods to implement its protocol. Therefore, all access to a given file remain strongly consistent (including read and write calls) even when the file is mapped in more than one domain and on more than one machine.

As mentioned before, the file system interposes a local attribute caching file system (CFS) agent at each local node. The CFS caches file objects implemented by remote nodes. Operations on remote file objects are redirected to the local CFS as described in [2]. Therefore, all read and write calls, as well as file attribute calls, are handled locally by the CFS. In addition, the CFS caches all file attributes from the remote file system and cooperates with the remote file system to keep them coherent using its own attribute coherency protocol.

An important function of the CFS is to handle bind requests for remote files. Recall that a bind request is necessary for each user map request. Using the CFS, all file operations including bind requests are handled locally.

Finally, the CFS utilizes the separation of the memory object from the pager object to ensure that all VMM page-in and page-out requests go directly to the pager object implemented by the remote file system, while at the same time handling all file (and memory object) operations locally. Reference [2] describes in detail the architecture and implementation of the file system.

9 VMM implementation

In this section we give a brief overview of the VMM implementation. We describe the overall organization of the system and highlight important structures. We elide many details that are not important for understanding the overall structure of the implementation.

As with other servers in Spring, the virtual memory manager is implemented in C++ in a multi-threaded fashion.

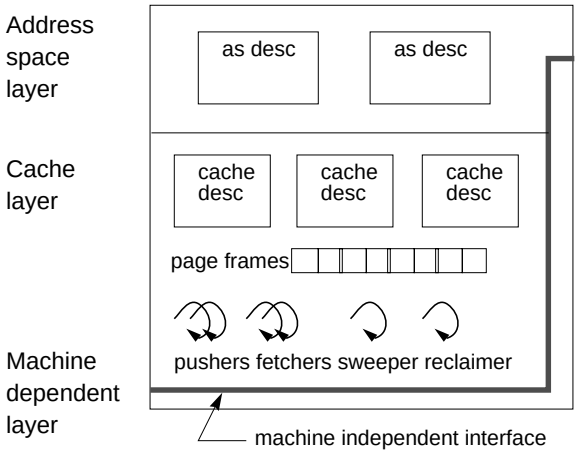

Figure 6. Internal structure of the VMM

The implementation is arranged in three layers: *address space*, *cache*, and *machine-dependent* layers as shown in Figure 6.

9.1 Address Space Layer

Each address space is represented by an *address space descriptor* structure that maintains a sorted doubly-linked list of *region descriptor* structures each of which describes a mapped region (Figure 7). A pointer to the most recently accessed region descriptor is kept in the address space descriptor. The address space descriptor structure also contains a mutex, a reference count, per-address space statistics, and per-address space swap pager and event handler objects. The address space descriptor also includes a *hat address space descriptor* that is used to manipulate the MMU translations for this address space (section 9.3). All address space descriptors are linked on a global list which is protected by its own mutex.

Each region descriptor contains the starting virtual address and length of the region described by the structure. The region descriptor also contains a pointer to a *cache descriptor* (section 9.2.1) and a list of *subregion map*s. Since each page in a region may have different protection and locking attributes, a subregion map is used to describe contiguous similarly mapped subregions. In practice, most regions have the same attributes for all their pages.

A reference to the memory object used in establishing a mapped region is kept in the region descriptor. This refer-

ence is necessary to guard against the situation where all references to the memory object are gone before the memory is unmapped. In general, the system informs the implementor of an object when all references to the object are gone. If a reference to the memory object is not kept by the VMM, the pager may mistakenly assume that the memory object is no longer in use and proceed to destroy the cache thus invalidating the mapping.

There is one specialization on the address space descriptor, *nucleus address space descriptor*, which adds a few extra methods for handling some of the special needs of the kernel.

9.2 Cache Layer

9.2.1 Cache Management

A cache descriptor is an abstract class that provides operations to page-in, page-out, attach/detach a region, and other operations related to maintaining physical pages in the cache. In addition to a mutex, it contains a condition variable that is signalled whenever a page belonging to this cache changes state (see below). A given cache descriptor is linked on one of several lists:

- attached cache list: A cache is linked on this list if it is currently mapped in some address space or if it is the source of a copy-and-write operation.
- new cache list*:* Caches that are newly created by the VMM from one of the swap servers are attached on this list. Caches are moved to the attached cache list when they are used.

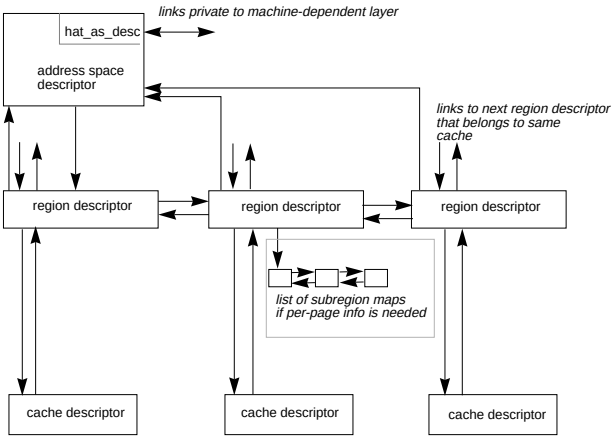

Figure 7. Address space and related structures

- special cache list*:* Special caches such as those used for mapping the kernel itself or for device support (see section 9.3.3) are attached on this list.
- unattached text cache list, unattached data cache list, unattached anon cache list: A cache is linked on one of these lists when (a) it is currently not mapped or used as a source of a copy-and-map operation, and (b) the pager that created it indicated that the cache should be kept around after the last attach to the cache is gone. The hint sent by the pager when the cache was created (section 5.1) determines the list used. Unattached caches are eligible for reclamation by the VMM.

There are several specializations on a cache descriptor. The most common is the *paged cache descriptor* which is used to implement instances of the VMM cache object. The paged cache descriptor encapsulate a pager object that is used to talk to the pager that controls the cache.

9.2.2 Physical Page Management

Each physical page is described by a page frame C++ object.[9] Each page frame has a number, a mode, a mutex, information about the cache to which it belongs, and state flags. A page frame belongs to at most one cache descriptor. At any point in time a page frame is linked on one of the following lists:

- free list: when the page is free and is not in use by any cache.
- in-use list: when page is in use in some cache. The list is arranged such that the least-recently used pages are at the front of the list.
- page-out list: when page is waiting to be paged-out to its pager object. The page is also in-use in some cache.

The page-frame lists are controlled by a global mutex. In addition, when a page is not on the free list, it is also linked on a per-cache list and a system-wide page hash table. The hash table is indexed by the <cache address, offset> pair. The hash table is used to locate pages given their offset within a cache (e.g., during the address translation process described in section 9.4). Figure 8 shows the various page frame lists.

9. Each physical page is actually a multiple of the MMU page. In the sun4c and sun4m implementations, a physical page is equal to the MMU page and is equal to 4K bytes. Only the machine dependent portion of the system knows about MMU pages.

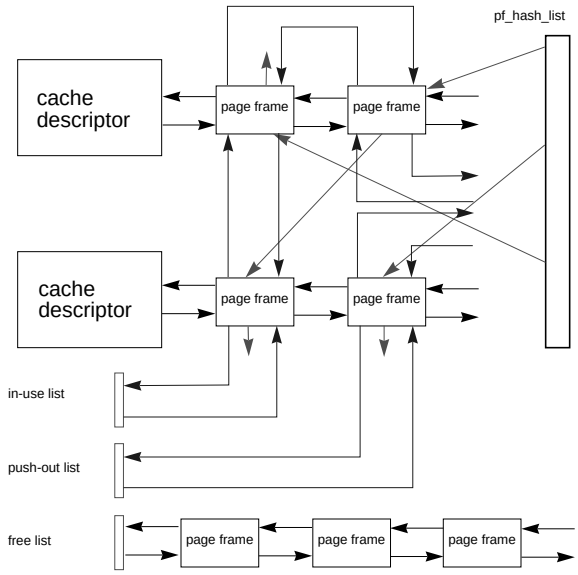

Figure 8. Data structures used for page frame linkage

A page frame may be *in-transition*. There are three types of transitions:

- page-in: page is in the process of being paged-in.
- page-out: page is in the process of being paged-out.
- other: page is in a temporary short transition.

If someone needs a page frame that is in transition, they have to release its lock and wait on the condition variable of the cache where this page frame belongs.

9.2.3 Internal threads

The cache layer employs several threads:

- Pushers. These threads are responsible for processing the push-out list.
- Prefetchers. Threads that are used to prefetch pages into memory.
- Sweeper. A thread that implements a two-hand clock-like sweep algorithm.
- Cache reclaimer. A thread that is responsible for deleting unattached caches. Any dirty pages are first flushed out before deleting an unattached cache.

9.2.4 Page Replacement

A global replacement algorithm is used in our current implementation. As mentioned before, the sweeper thread uses a two-hand clock algorithm to keep track of referenced pages. The system maintains several counters of clean and dirty pages and uses them to schedule dirty pages to be cleaned when it is running out of clean pages.

9.3 Machine-dependent Layer

The machine-dependent layer encapsulates all knowledge of the MMU and hardware caches. This layer has a machine-independent interface that is used by the rest of the system.

9.3.1 Machine-independent Interface

There are two C++ classes that define machine-independent interfaces for maintaining the MMU and the hardware caches. These classes are used to enter and remove translations from the MMU.

An object of type *hat address space descriptor* is included as part of each address space descriptor. This object defines an interface for manipulating the address space MMU translations. Operations include entering the translation for a page frame at a particular virtual address, removing a translation, and changing the access mode of an address range.

An object of type *hat frame descriptor* is included as part of each page frame. This object defines operations for manipulating MMU translations to this particular page frame. The hat address space descriptor and hat frame descriptor maintain the relationship between each page frame and the address space descriptor(s) where the page frame is currently mapped. This information is used when changing the MMU translations of page frames.

9.3.2 Machine-dependent Code

The implementation of the machine-dependent layer maintains three types of information:

- The relationship between each page frame and the address space(s) where it is currently mapped.
- Any MMU-specific state information (e.g., hardware-dictated page tables for MMU's that require such tables).
- A cache of virtual-to-physical memory translation information (the "software TLB" in Figure 9). This state is only a cache and the implementation is permitted to "forget" about a translation since the true state of virtual-to-physical and physical-to-virtual translation

information is maintained by the address space and cache layers.

The current implementations of the hat address space descriptor and hat frame disc classes use the hardware address translation (*HAT*) layer of SunOS [13]. The implementation of these two classes consists mainly of direct calls to the HAT layer routines.

Using the SunOS hat layer has the advantage of fast bring-up on existing and future machines. No changes where made to the hat layer itself with the exception of adding a few #ifdefs and #defines.

9.3.3 Device Support

There are two specializations on the class *cache* descriptor that are machine-dependent. Class *device register cache* descriptor is used in the implementation of *device register memory object*. Similarly, *dvma cache* descriptor is used in the implementation of *dvma memory object*.

Note that to port the virtual memory system to a different architecture or MMU, only the implementations of hat address space descriptor and hat frame descriptor classes and any necessary device support need be changed (see section 9.7).

9.4 Address Translation Process

Figure 9 summarizes the address translation process. Virtual addresses used by instructions executed by a thread running in a domain are translated to physical addresses using the hardware address translation cache. If the hardware cannot translate a virtual address, a machine translation fault is generated. A machine-dependent handler attempts first to locate the required translation in a software cache that acts as an extension to the hardware TLB. If the translation is not found, the following steps are taken:

1. The trap handling code locates the currently running domain and invokes the *handle_page_fault* method on its address space descriptor object.
2. The address space layer locates the region descriptor where the fault occurred, checks protections and invokes the page_in method on the cache descriptor that backs the region where the fault occurred.
3. The cache layer attempts to locate the required page. Depending on the particular cache descriptor type this process may involve a simple lookup in a linear table or a more sophisticated search. We describe here the implementation of the paged cache descriptor, the most common (and complex) implementation of cache descriptor. The search is started by first looking in the source copy-on-write cache, if any (section 9.7), then searching for the required page in the system-wide page hash table, and finally in the per-cache zero-fill table. If the page is located, it is returned to the caller, otherwise the page_in method of the pager object corresponding to the cache object is invoked. When the call returns, the page is returned to the caller.
4. The address space layer then invokes the appropriate call on its hat_address space descriptor object to enter a translation for the page and returns.
5. The trap handler returns, retrying the faulting instruction.

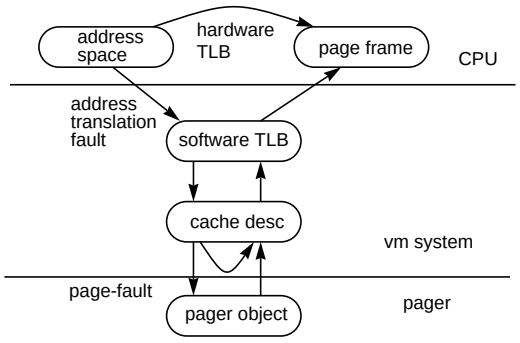

Figure 9. Address translation process

Virtual to physical address translations are normally cached in the hardware TLB. On an address translation fault the VMM is invoked. The cache responsible for backing the virtual address where the fault occurred is consulted. The cache may service the translation request by returning an existing page frame or it may contact a pager object to page-in the required page.

When the page_in method on the cache descriptor is called, both the address space descriptor and cache descriptor locks are held. If a page-in request to the pager has to be issued, both of these locks must be released first since the call is an out-call that may take an arbitrarily long time to return. Not releasing the locks introduces deadlock, security, and performance problems. Therefore, the reference counts on the address space descriptor and cache descriptor are incremented, a stub page frame is entered in the paged cache descriptor and locks are released. (The purpose of the page frame stub is to indicate that an operation on the page is in-progress so that another request for the same page will wait instead of attempting another page-in.) When the call returns the locks are re-acquired, a check is made to see whether the address space

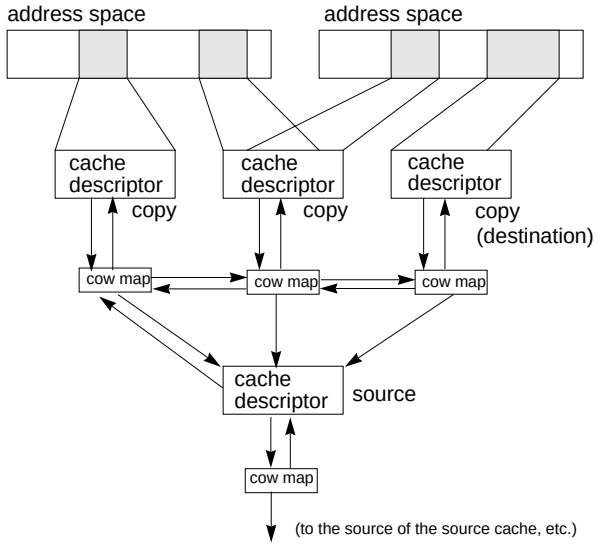

Figure 10. Copy-on-write data structures

The figure shows three cache descriptors that are copies of a source cache. The source cache is in turn a copy of another cache (not shown). Some of the cache descriptors back mapped regions in two address spaces.

descriptor or cache descriptor has been (logically) deleted and the page frame stub is removed from the cache.

Note that the fault handling code path is the most frequently executed part of the virtual memory. Therefore, the order of acquiring internal locks is designed such that it is the natural order for this code path.

9.5 Copy-on-write implementation

The VMM implementation provides copy-on-write (COW) support that is used to implement the *copy_and_map* method of the address space object (Appendix A), and is used in bulk data transfers with copy semantics (section 9.6). COW support is part of the paged cache descriptor implementation.

9.5.1 Data structures

The COW data structures maintained by the implementation are depicted in Figure 10. A given (destination) cache may be a copy of a portion of another (source) cache. Each destination cache has a copy-on-write map structure (*COW map*) that contains information about the logically copied range. This information include pointers to the source and destination caches, the starting offsets in the source and destination caches, length of the copied range, and a bitmap indicating which pages have been copied from the source cache. The COW map also keeps a reference to the memory object that is backed by the source cache. This reference is kept for the same reason as the one explained in section 9.1 for a region descriptor. The source cache in turn keeps a linked list of all COW maps that point back to it. The need for this list is explained below in section 9.5.4.

It is possible to have a chain of caches that are copies of other caches. This structure can nest to arbitrary levels.

9.5.2 Setting up a COW relationship

The following steps are taken when a copy-on-write relationship is established:

- A COW map structured is allocated and initialized. The structure is linked as shown in Figure 10.
- A new region in the destination address space is mapped. This region is backed by the destination cache.
- The MMU-level access mode of each region that maps the source cache in read-write mode is changed to read-only. This is to catch attempts to modify the source cache and to trigger COW processing as explained in section 9.5.4. Note that only the MMU-level access mode is changed; the true access mode in the region descriptor remains the same.

When a copy-on-write relationship is established, pages at the source and destination caches are not copied until a write is made to either the source or the destination page. Therefore, the copy-on-write data structures are consulted at two points during fault processing: at the beginning of a page-in request when trying to locate a page, and at the end of a read-write page-in request to update any read-only destination caches. Note that the page-in is a request to the cache and as mentioned before in section 9.4 does not necessarily mean an actual page-in from the pager.

9.5.3 COW processing at the beginning of page-in

When handling a page-in request, the cache checks to see if it has a COW map attached. If there is no such structure, then copy-on-write processing ends and fault processing resumes as described in step 3 on page 16. Otherwise, the following is done:

- The COW map is queried to see if the required page falls in a range logically copied from another cache and the page has not been copied from the source cache

yet. If not, COW processing ends and fault processing resumes.

- Next, a check is made if the fault is a read-only fault *vs.* a read-write fault.
- For a read-only fault, a page-in request is issued to the source cache. The page returned from the page-in is used to satisfy the fault and no copy is made (i.e., it is shared read-only between the source and destination caches).
- Otherwise, an internal copy-to request is issued on the *source* cache to make a copy of the page and return it.
- As part of processing the copy-to request, the source cache first issues a page-in to itself (which may initiate COW handling on the source cache), then returns a copy of the page. The COW map is also appropriately updated to indicate that this particular page has been copied from the source cache.

9.5.4 COW processing at the end of page-in

As shown in Figure 10, a given cache may be the source to a number of destination caches. Before a page-in on a cache ends, the following checks are made:

- If the page-in request is for a read-write page,
- and if the cache is a source cache to any other caches,
- and if there are any caches that have not yet made their own copies of this page,

then the linked list of all destination COW maps is traversed (Figure 10), and a copy of the page is made for each destination cache. Note that this copy must be made at this point to maintain the COW semantics.

9.5.5 Tearing down COW data structures

The implementation takes considerable care in building and tearing down the COW data structures shown in Figure 10. Careful reference counting and locking is needed to maintain these structures.

Each COW map linked to a source cache is considered an attachment to the cache (similar to an attached region descriptor as in Figure 7). As long as a cache is still attached, it is not subject to reclamation by the VMM. However, a cache that is the *destination* of a copy is not considered attached unless it is also attached to a region.

Although the VMM will only reclaim unattached caches, a cache may be destroyed at any point by the pager. Therefore the implementation has to guard against either the source or destination disappearing while accessing the structures.

9.6 Bulk data transfer

The address space layer cooperates with the cache and the machine-dependent layers to provide the nucleus with bulk data transfer support during object invocation. Before we present the data transfer mechanism, we briefly describe the object invocation mechanism.

9.6.1 Spring Object Invocation

An object invocation as seen by the Spring nucleus is a trap from user-mode that presents the nucleus with a door (handle) id to invoke, plus optionally some arguments to pass to the domain that implements the object represented by the door id. A return from an object invocation is very similar, except that in this case the return address is known by the kernel and is not specified by a door id.

The arguments to the call may consist of a limited number of words that fit in machine registers or a pointer to a *transport buffer.* A transport buffer may have up to three parts:

- A set of door ids.
- In-line data
- Indirect data.

The door ids are passed to the destination domain by the nucleus. The in-line data is copied directly by the nucleus from the transport buffer to the destination domain. The nucleus uses the address space layer to pass the indirect data to the destination domain.

Indirect data may be passed either by *copy* or *move* semantics. Note that the transport buffer is normally set up by stub and library routines in user-mode before the call is made. These routines determine whether to pass the data in-line, by copy, or by using the move semantics.

For each indirect data block, the transport buffer includes an indication of whether to copy or move the data. Copying the data is done using the copy-on-write mechanism and has the expected semantics. The semantics of moving the data is equivalent to unmapping the memory from the source domain and mapping it in the destination domain.

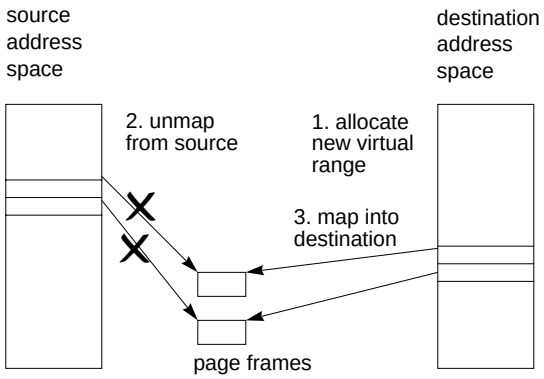

Figure 11. Moving pages during object invocation

9.6.2 Implementation of the *move* operation

The implementation is optimized such that, in the common case, the data transfer is reduced to invalidating the translations in the source address space and setting up the new translations in the destination address space to the *same* physical pages.

A bulk data transfer is initiated when the nucleus issues an internal transfer call to the virtual memory system. The request specifies the source and destination address spaces, a range of addresses in the source address space, a length, and an indication of whether to copy or move the data. If a copy is requested, the VMM establishes a normal copy-on-write mapping between the source and destination address ranges. For a move request, the following steps are taken (Figure 11):

- A range of free virtual addresses is allocated in the destination address space. The backing store of this range is the default memory object of the destination address space. This is a relatively fast and simple operation.
- For each page in the source address range, a check is made if the page is resident in memory and if the necessary locks can be obtained. If this test succeeds, the page translation is invalidated from the source address space, a new translation is entered into the destination address space, and the identity of the page is changed to the destination cache descriptor. If the test fails, a more general (and slower) code path is taken.
- The general code path handles the cases of non-resident or in-transit pages, and waits for any necessary locks. The implementation is careful to guard against deadlocks while obtaining the necessary locks.

Huge amounts of data can be transferred using this mechanism. The amount is only limited by the size of the *virtual* address space. Any or all of the data being transferred could be paged out of memory.

Note that the current implementation of the fast path data transfer is machine-independent. Further optimizations are possible, but a faster implementation requires coding short sequences of machine-dependent assembly language, and more importantly, knowledge of underlying memory management hardware (section 9.3).

9.6.3 Example

An important use of the bulk data mechanism is to move data efficiently between stable storage and the virtual memory system. In Spring, the disk driver, the pager, and the VMM may reside in separate domains. Therefore, the data may have to be transferred between several domains during a paging operation. We would like to maintain the flexibility of placing the various servers involved in paging data in different domains, and we do not want this decision to be constrained by the cost of moving data between the different servers.

We present an example that illustrates how the bulk data transfer mechanism is used to efficiently move data during a page-in operation. Figure 12 shows three servers that are involved in paging data from a disk drive: the disk driver, a pager, and the VMM. The following steps correspond to the ones in the figure:

1. VMM issues a page-in request to the pager.
2. The pager in turn issues a disk read request to the disk driver.

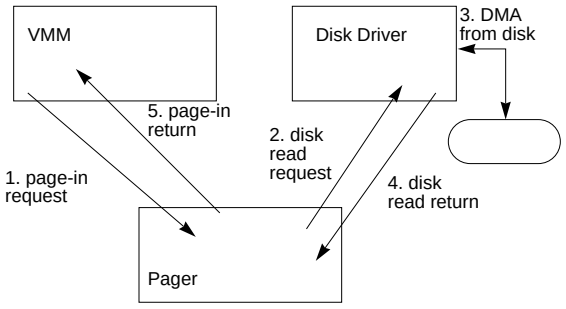

Figure 12. Paging from a local disk

3. The disk driver allocates memory and initiates a DMA (direct memory access) operation into the allocated memory.
4. The disk driver returns the memory using the bulk data transfer mechanism. As part of the return from the disk read operation, the memory is unmapped from the address space of the disk driver and is mapped into the address space of the pager as explained before in section 9.6.2. No data copying takes place.
5. The pager returns the required data to the VMM, again using the bulk data transfer mechanism. The VMM uses the returned pages to satisfy the page-in request.

9.7 Implementation Status

All functionality described in this paper has been implemented and is part of the base Spring system. Our initial implementation was for the *sun4c* architecture (SPARCstation 1, 2). The system was then ported to the *sun4m* multiprocessor architecture (SPARCstation 10 and SPARCserver 600) without modifying any of the machine-independent portions of VM—only the machine-dependent portion described in section 9.3.2 was changed. Note that the *sun4c* and *sun4m* architectures have drastically different MMUs. All system servers including the VMM are multi-threaded, and the system runs in uniprocessor and symmetric multiprocessor configurations.

There are several pager implementations, the most important of which is the Spring file system. The system interfaces are stable; some of the interfaces, especially the cache-pager object interface, underwent several revisions as we gained more experience with the system.

The Spring system as a whole is now very stable and usable. We are starting to use it with the X11 window system as a development environment.

9.8 Testing

We used several approaches in testing the implementation. In addition to ad-hoc stress testing (both synthetic stress tests as well as compiling the system using itself), we wrote a test suite to pseudo-randomly execute various virtual memory operations. The pseudo-random test has been successful in finding bugs at boundary conditions and in executing complex mapping and unmapping operations. It is less useful as a stress test but very good at generating unexpected legal (and illegal) test sequences that a test writer would not normally code. Finally, as an ongoing activity, we are annotating the interfaces with semantic clauses using a formal specification language with an eye toward automatically generating test programs from the specifications.

9.9 Performance Evaluation

Work in this area has concentrated on three approaches:

- Instrumenting with trace records. A circular buffer maintains event traces that can be dumped (from a running or a stopped system) and analyzed off-line.
- Dynamically observing the system. The virtual memory system exports various dynamic information that is displayed and observed graphically. Each address space provides usage statistics as well as mapping information, while the system as a whole provides various statistics including paging activity, fault rates, and cache activity.
- Statistically observing the system. Basic *gprof* support is implemented in Spring and it is used to obtain statistical profiles of the system.

We have done some performance tuning and measurements. The implementation is efficient. For example, the virtual memory system is able to achieve a paging throughput in excess of 95% of raw system io throughput.

More performance evaluation and tuning are needed in some areas, for example, page replacement. We are in a position to evaluate and tune page replacement now that we have a complete system with a set of real applications.

10 Conclusions and Future Extensions

We have designed and implemented a virtual memory system for a general-purpose operating system that emphasizes object-oriented interfaces, security, and distribution. The virtual memory system separates the memory abstraction from the interface that provides the actual memory and provides an architecture for efficient sharing of the underlying memory in a secure manner.

It is interesting to note that one aspect of our original design that we revisited time and again during the implementation is the cache-pager object interface. Although the resultant interface does not differ much from the one

we initially designed, we continually made small changes to this interface during the development effort.

The implementation of the system described in this paper is complete and is currently in use. We are currently undertaking an effort to tune the system, including measuring and tuning lock usage on multiprocessor systems. Also we are starting to experiment with various paging and cache reclamation policies. In addition several extensions to the system are planned:

- The address space object will be extended by adding two calls: an *advise* call that gives the VMM hints of the expected behavior of an application, and a *status* call that returns and clears information regarding modified and referenced pages of the address space. The status call is useful for garbage collectors.
- The behavior of the VMM when time-outs and failed invocations occur will be specified more precisely.
- Page frames of only one size are supported. With the advent of MMU's that support multiple page sizes and of machines that support giga-bytes of physical memory and tera-bytes of virtual memory we believe there is a need to support multiple sized pages.

References

[1] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of Winter '93 USENIX Conference*, January 1993.

[2] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany, "The Spring File System," Sun Microsystems Laboratories, SMLI-92-389, December 1992.

[3] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, 37(8):896-908, August 1988.

[4] Michael Wayne Young, "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System," Technical Report, CMU-CS-89-202, Carnegie Mellon University, November 1989.

[5] Vadim Abrosimov, Marc Rozier, and Marc Shapiro, "Generic Memory Management for Operating System Kernels," *12th Symposium on Operating Systems Principles (SOSP '89)*, pp. 123-136, 1989.

[6] Vadim Abrosimov, Francois Armand, and Maria Inés Ortega, "A Distributed Consistency Server for the CHORUS System," *3rd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pp. 129-148, March 1992.

[7] José M. Bernabéu-Aubán, *et al.,* "The Architecture of *Ra*: A Kernel for *Clouds*," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, pp. 936-945, January 1989.

[8] Kieran Harty and David R. Cheriton, "Application-Controlled Physical Memory using External Page-Cache Management," *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 187-197, September 1992.

[9] David R. Cheriton, "The Unified Management of Memory in the V Distributed System," Technical Report CSL-TR-88-359, Computer Science Laboratory, Stanford University, August 1988.

[10] E. I. Organick, *The MULTICS System: an Examination of its Structure* (1972). MIT Press, Cambridge, Mass.

[11] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf, "The architecture of an integrated local network," *IEEE Journal on Selected Areas in Communication*, SAC-1(5):842-57 (November 1983).

[12] Vincent Russo and Roy H. Campbell, "Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design," *Proceedings of OOPSLA '89*, pp. 267-278, September 1989.

[13] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, "Virtual Memory Architecture in SunOS," *Proceedings of Summer '87 USENIX Conference*, June 1987.

Appendices

The following appendices list the interfaces of the major virtual memory objects. Appendix A lists the interface of the address space object. Appendix B lists the interface of the cache object which is exported by the cache manager interface (the VMM acts as cache managers). Appendix C lists the vmm object interface, while the interfaces of the memory and pager objects are listed in Appendix D.

The code below specifies for each parameter a passing mode: a Spring object passed *copy* remains accessible to the caller and callee after the call is made, while a *consume*d object is deleted from the calling domain as a side effect of the call. *Borrow* is an in-out passing mode, while *produce* is an out mode. Due to space considerations we elide some methods, most comments, and type declarations. Most methods raise exceptions when errors are encountered; we elide the description of the exceptions as well.

A Address space object interface definition

```
interface address_space {
    void map(
        copy memory_object mobj,
        copy offset_t mobj_offset,
        borrow offset_t length_in_bytes,
        borrow addr_t as_address,
        copy access mode
     );
    void map_many(
        borrow mappings_list mappings
    );
    void unmap(
        copy addr_t address,
        copy offset_t length_in_bytes
    );
    void copy_and_map(
        copy memory_object source_mobj,
        copy offset_t source_mobj_offset,
        copy offset_t length_in_bytes,
        borrow addr_t as_address,
        copy access mode,
        produce memory_object destination_mobj
    );
    void allocate_memory_and_mobj(
        borrow offset_t length_in_bytes,
        borrow addr_t address,
        copy access mode,
        produce memory_object new_mem_obj
    );
    void allocate_memory(
        borrow offset_t length_in_bytes,
        borrow addr_t address,
        copy access mode
    );
    void change_access_rights(
        copy addr_t address,
        copy offset_t length_in_bytes,
        copy access new_access_rights
    );
    void flush(
        copy addr_t address,
        copy offset_t length_in_bytes,
        copy boolean please_keep
    );
    long get_page_size(
    );
    void get_mappings(
        copy addr_t address,
        copy offset_t length_in_bytes,
        produce mappings_list mappings_description,
        produce boolean more_regions
    );
    void statistics(
        produce as_statistics stats
    );
    void add_swap_pager(
        copy swap_pager swapper
    );
    void lock_memory(
        copy addr_t address,
        copy offset_t length_in_bytes
    );
    void unlock_memory(
        copy addr_t address,
        copy offset_t length_in_bytes
    );
    void catch_as_events(
        copy as_handler handler
    );
}; // address_space
```

B Cache object interface definition

```
interface cache_manager {
    void create_cache_object(
        consume pager_object cache_pager,
```

```
            copy boolean signal_last_ref,
            copy pager_info info,
            copy ownership_t vcache_ownership,
            produce vcache_object new_cache,
            copy rights maximum_access_rights,
            produce rights_list cache_rights_list
        );
        void create_cache_object_and_bind(
            consume bind_key_object bind_key,
            copy offset_t bind_length,
            copy offset_t bind_cache_offset,
            copy rights bind_max_rights,
            consume pager_object cache_pager,
            copy boolean signal_last_ref,
            copy pager_info info,
            copy ownership_t vcache_ownership,
            produce vcache_object new_cache,
            copy rights maximum_access_rights,
            produce rights_list cache_rights_list
        );
        void bind_cache(
            copy bind_key_object bind_key,
            copy offset_t bind_length,
            copy offset_t bind_cache_offset,
            consume rights_object rights_to_cache
        );

    }; // cache_manager

    interface cache_object {
        // The size_in_bytes argument can be specified as -1
        // to indicate all blocks starting from cache_offset to
        // the end of the cache.
        void flush_back(
            copy offset_t cache_offset,
            copy offset_t size_in_bytes,
            borrow ownership_t vcache_ownership,
            produce data memory_bytes
        );
        void deny_writes(
            // same parameters as flush_back()
        );
        void write_back(
            // same  parameters as flush_back()
        );
        void delete_range(
            copy offset_t cache_offset,
            copy offset_t size_in_bytes,
            borrow ownership_t vcache_ownership
        );
        void zero_fill(
            // same parameters as delete_range()
        );
        void populate(
            copy offset_t cache_offset,
            copy offset_t size_in_bytes,
            copy rights requested_access,
            borrow ownership_t vcache_ownership,
            copy data memory_bytes
        );
        void destroy_cache( );

    }; // cache_object interface
```

C Vmm object interface definition

```
interface vmm : cache_manager {
    void get_vm_info(
        produce vm_info local_vm_info
    );
    void add_swap_pager(
        consume swap_pager swapper,
        copy boolean temporary
    );

}; // vmm
```

D Pager objects interface definitions

```
interface memory_object {
    void bind(
        copy name cache_manager_name,
        copy rights requested_access,
        copy offset_t mem_obj_offset,
        borrow offset_t length_in_bytes,
        produce rights_object rights_to_cache,
        produce offset_t cache_offset,
        produce long flags
    );
    void final_bind(
        consume bind_key_object rights_to_bind,
        copy name my_name,
        copy rights request_access,
        copy offset_t mem_obj_offset,
        copy offset_t length_in_bytes
        produce long flags
    );
    void get_length(
        produce offset_t length_in_bytes
    );
    void set_length(
        copy offset_t new_length_in_bytes
    );

}; // memory_object interface

interface pager_object {
    void page_in(
```

```
            copy offset_t cache_offset,
            copy offset_t size_in_bytes,
            copy rights requested_access,
            borrow ownership_t vcache_ownership,
            produce data memory_bytes
        );
        void page_out(
            copy offset_t cache_offset,
            copy offset_t size_in_bytes,
            borrow ownership_t vcache_ownership,
            copy data memory_bytes
        );
        void write_out(
            // same parameters as page_out()
        );
        void sync(
            // same parameters as page_out()
        );
        void zero_fill_range(
            copy offset_t cache_offset,
            copy offset_t size_in_bytes,
            borrow ownership_t vcache_ownership
        );
        void done_with_cache(
            borrow vcache::ownership_t vcache_ownership
        );
        // done_with_pager_object is called by cache
        .// manager when it reclaims this cache.
        void done_with_pager_object(
        );
};  // pager_object
```