Multiple Coherency Models in XMM: Architecture and Design[1]

Michelle Dominijanni

Open Software Foundation
Research Institute
1 Cambridge Center
Cambride, MA 02142

Tel: (617) 621-7313
E-mail: mmp@osf.org

Abstract. The current version of XMM (eXtended Memory Mangement) supports a single coherency model: strong coherency, in which a page may be mapped read-only by multiple kernels, but only a single kernel may have the page mapped if it is writable. While this is precisely what is needed to support shared memory on a multicomputer, it provides more consistency than is needed on NUMA architectures, where the hardware maintains consistency, or for uses such as mapped files, where consistency is maintained at a higher level. This paper will describe the modifications needed in XMM, the Mach VM susbsystem, and the EMMI (External Memory Manger Interface) to support multiple coherency models simultaneously.

1. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Material Command (AFMC).

1. Introduction and Background

The XMM (eXtended Memory Management) subsystem in the NORMA microkernel provides distributed shared memory support in a multicomputer whose nodes do not have any physically shared memory. The XMM module resides between the VM subsystem and the memory manager(s). Since the original goal of XMM was to maintain the consistency of shared memory in a NORMA system, XMM was implemented with a single coherency policy, a strong coherency model. In this model, a page may be mapped read-only by multiple kernels simultaneously, but only a single kernel may have the page mapped if it is writable. XMM interacts with both the memory manager and the VM subsystem to guarantee this consistency.

However, strong consistency is not always needed or desired. The OSF/1 AD mapped file implementation uses memory objects for file data caching. While XMM does maintain the consistency of the file data, this is essentially wasted effort since the mapped file tokens are also used to enforce a single writer policy. The mapped files module actually has more information than XMM does about the file data, and it can manage the consistency more efficiently. Therefore it is desirable to have XMM support both the current strong coherency for anonymous shared memory, as well as a “no coherency” policy for mapped files.

Another instance where XMM provides more consistency than is needed is found on NUMA architectures. In this case, the physical memory might be accessible from multiple nodes and the hardware maintains the consistency, eliminating the need for enforcing a single writer policy. Still, XMM does or could provide a means to pass mapping information and data from kernel to kernel without having to contact a pager, as well as a convenient place to manage invalidation of mappings. From these needs arise both a need for another coherency policy and some extra functionality that logically belongs in XMM.

Since much of the motivation for allowing multiple coherency policies in XMM comes from the need to support architectures other than NORMA, some of the work outlined in this design is not strictly related to breaking out the policy-specific parts of XMM. However, these extensions are necessary for correct and/or efficient use of NUMA architectural features and will not adversely affect current usage.

The code base for this discussion is NMK 14.10.

2. Design Overview

The first part of the design will address the support of multiple coherency policies. This first requires the definition of an interface for the policy choice to be indicated to XMM. Once this interface is defined, the users of the interface need to be identified and changed to use it. These items will be discussed in the context of the initialization of the XMM stack.

The next section will discuss the addition of a mechanism in XMM to allow the existence and use of multiple coherency policies. The current strong policy code will be broken out as the first policy.

The rest of the design describes extensions to XMM and the External Memory Management Interface (EMMI) to allow them to support and take advantage of NUMA architectural features. In particular, data copying can be minimized in such a system by passing around physical addresses instead of pages of out-of-line data. To do this, first a new kernel-internal representation of a page in this format will be defined. Next, the pagein path will be altered slightly to allow a page to be preallocated and its physical address given to the pager. The pager can then set up a transfer from disk directly into the target page.

Finally, with a relaxed coherency policy that allows multiple writers to map a page, it becomes necessary to define a mechanism for the pages to be made inaccessible prior to a pageout. This mechanism will not be made specific to the coherency policy but will instead be implemented as an option to be explicitly requested upon pageout.

3. Multiple Coherency Policy Support

Support for multiple coherency policies first requires that there be a place for a coherency type and its values to be defined. Since the coherency policy needs to be visible to both the kernel and the memory manager, the logical place for this definition is in mach/memory_object.h which contains similar definitions for things such as copy strategy.

```
#if NORMA_VM

typedef int memory_object_coherency_t;

#define MEMORY_OBJECT_COHERENCY_STRONG        1
#define MEMORY_OBJECT_COHERENCY_NONE          2

#endif NORMA_VM
```

Next, the coherency needs to be stored within XMM. For the current policies, this only needs to be within the SVM module although nothing would preclude other modules also storing the policy. A member called coherency of type memory_object_coherency_t needs to be added to struct mobj in norma/xmm_svm.h.

3.1 Initialization

This section describes the method by which the coherency policy is initially specified. No assumptions are made about how the VM susbsystem determines the particular policy to be used; one possible source of this information might be the VM mapping. It makes more sense to derive the coherency policy from something like the VM mapping or VM object and then set up the XMM stack accordingly than to defer the policy choice until the memory manager has a chance to reply to the initialization sequence, since this gives more flexibility to the policy implementor. While it is not the intent at this time to change the XMM stack itself for the different policies that are going to be implemented, it is conceivable that some new policy might want to replace one of the XMM modules and not just the policy portion of the SVM module. This is made possible by specifying the policy at initialization time.

There will be no mechanism to change the policy once the XMM stack has been created. Doing this would require that there be no outstanding or in-progress operations on the memory object, and probably that there be no resident pages from the memory object on any kernel. Achieving this state would be almost equivalent to destroying the memory object and would probably require some fairly complex changes to both XMM and VM. A much simpler method to change the policy if necessary is to destroy and recreate the memory object.

The XMM stack is created at the time the VM object is set up by vm_object_enter() (vm/vm_object.c), with a call to xmm_memory_object_init(). This is currently an asynchronous operation, assumed to always succeed. This presents a problem in the case where one kernel has already set up the XMM stack and specified a policy, and a second kernel tries to specify a different policy. The second attempt must fail, and this requires both that xmm_memory_object_init() return success or failure and that vm_object_enter() and its callers (specifically vm_map()) be prepared to handle the failure.

First, a memory_object_coherency_t argument needs to be added to xmm_memory_object_init() (norma/xmm_server.c). If this function is going to fail because of an incorrect coherency, this should be detected at the earliest opportunity so that a minimum of data structures and port rights need to be deallocated on failure. This is in xmm_object_by_memory_object() (norma/xmm_object.c), which is where the memory_object port is associated with an xmm_object port and SVM mobj structure (either of which may be remote). Accordingly, the coherency object must also be passed to xmm_object_by_memory_object(). Currently, xmm_object_by_memory_object() is assumed not to fail. The assertion to that effect in xmm_memory_object_init() must be removed and replaced with return KERN_FAILURE (or some other more appropriate error, if one exists) if xmm_object is IP_NULL.

Next, xmm_object_by_memory_object(), xmm_object_by_memory_object_remote(), r_norma_xmm_object_by_memory_object(), and norma_xmm_object_by_memory_object(), all in norma/xmm_object.c, and the definition of norma_xmm_object_by_memory_object() in norma/norma_internal.defs must all be modified to include a coherency argument. The substantive modifications are all in xmm_object_by_memory_object() since the rest are all part of the mechanics of getting the request to the proper node. In xmm_object_by_memory_object(), there are three possible paths:

1) an xmm_object already existed,

2) a new xmm_object is created but someone else created one in the meantime (the race was lost), and

3) a new xmm_object is created.

In case (3), xmm_svm_create() (norma/svm_temporary.c) should have a coherency argument added, and it should be modified to set the coherency. In cases (1) and (2), a new routine, xmm_svm_coherency_get(), should be called to get the already existing policy to compare it to the policy passed in as a parameter. If they don’t match, then one send right for the xmm_object should be discarded and IP_NULL should be returned. The new routine will require the SVM mobj as an argument, which is obtained from xmm_object->ip_kobject.

The new routine should be added to norma/svm_temporary.c and should be declared:

```
memory_object_coherency_t
xmm_svm_coherency_get(mobj)
xmm_obj_t mobj;
{
        /* return MOBJ->coherency */
}
```

Getting back to vm_object_enter(), we now have a call to xmm_memory_object_init() that can potentially fail. The return value from xmm_memory_object_init() must now be examined, and if it failed, then appropriate deallocations must be done (see next paragraph) and VM_OBJECT_NULL returned. This new failure mode for vm_object_enter() can only occur on behalf of an external user, since other callers of vm_object_enter() are creating shadow or copy objects, or represent internal objects, all of which the kernel has control over. Thus, vm_map() is the only caller that must be able to handle the new failure of vm_object_enter(), and it is already capable of doing so.

As to the specific actions that must be taken in vm_object_enter() upon failure of xmm_memory_object_init, there are two paths that need to be addressed. The first is the thread that is actually doing the initialization, and the second is another thread that is waiting for the initialization to be finished. The sequence of events in the initialization is to set pager_created to TRUE, record the pager port, contact the pager, then set pager_initialized to TRUE and wake up anyone who is waiting. Currently, there are two places that wait for the initialization: vm_object_deallocate(), which blocks if pager_created is TRUE and pager_initialized is FALSE, and vm_object_enter() itself, which blocks waiting for pager_initialized to become TRUE. The correct action for vm_object_deallocate() is to continue when the action of contacting the pager is no longer in progress, whether it is successful or not, so if pager_created is set to FALSE upon failure of xmm_memory_object_init(), then vm_object_deallocate() will already proceed when awakened. The while loop near the end of vm_object_enter() which blocks waiting for the initialization to finish needs to be modified to check for pager_created being set to FALSE; if this is detected, then the object reference should be dropped and the function restarted (a restart tag already exists). Finally, if the call to xmm_memory_object_init() returns failure, the following actions should be taken: set pager_created to FALSE and pager to IP_NULL, release a send right on the pager port, release an object reference, call vm_object_wakeup() (as is done later in the routine), and return VM_OBJECT_NULL.

While the memory object initialization sequence just described does not depend on it, a final change that should be made is to add a coherency argument to memory_object_init(), and to its XMM counterparts, M_INIT() and the module-specific functions m_*_init(). This change is desirable for several reasons:

1) this enables a synchronous exchange with the pager to allow it to reject or negotiate the policy choice;

2) it allows any module within XMM to keep track of the policy choice if it wishes to make use of the information;

3) this is a simple change to the EMMI that is best made along with the other EMMI changes proposed in this design.

3.2 Separation of Policy Implementation and Multiple Policy Support

In order to have more than one policy in XMM, it is necessary to identify the policy-specific portions of the existing code and add an interface between the policy-specific code and the policy-independent code. One approach that could have been taken was to completely replace the svm_process_request() the function that implements the state machine in norma/svm_request.c. The chief benefit to this approach is that it would provide the greatest flexibility to the policy implementor. However, experience with the existing code has proven the difficulty of getting the state machine implemented correctly, and a study of svm_request.c shows that the policy specific code is contained almost entirely in one routine that is indirectly called from the state machine. Changing the state machine would most likely require changes to much more of svm_request.c than really necessary, thus greatly increasing the risk of error.

The alternate approach is to identify exactly which routine(s) are policy-specific, and then provide a pointer in the mobj to an operations vector so that the policy-specific routines may be called based on the coherency policy also stored in the mobj. This results in a much simpler implementation effort that is easily extensible to include more routines should the need arise in the future.

To this end, a pointer to an operations vector should be added to the svm mobj structure. This pointer should be initialized at the time the policy is set to point to the appropriate vector of policy-specific operations. A new file should be created, svm_coherency.c, which will contain these vectors as well as the policy-specific routines. A declaration of the operations vector itself should be placed in another new file, svm_coherency.h, so that it can be included by svm_request.c, svm_temporary.c, and svm_coherency.c.

One of the consequences of creating the new files is that most of the structure declarations in svm_request.c will need to be moved the svm_request.h header file so that they are accessible to svm_coherency.c.

A pass through svm_request.c shows that the policy-specific code is almost entirely contained in svm_start_kernel_operation(). The current implementation of this function maintains a single writer for a page by generating lock_requests to kernels other than the requesting kernel. This will become the first policy-specific routine, and as such should be moved to svm_coherency.c and renamed to strong_start_kernel_operation().

There is one other function that contains policy-specific information: svm_check_dirty_invariant(). This is a debug routine that is called in many places throughout svm_request.c to check consistency of the svm data structures and enforce the strong coherency policy. It is conditionally defined under #if MACH_ASSERT. It would not be advisable to remove this routine entirely since it has been very useful in tracking down problems in SVM; however, it will obviously not work with policies other than strong coherency. It is likely that a similar consistency checking routine would be useful for other policies, so this should become one of the coherency policy operations and therefore be moved to svm_coherency.c and renamed to strong_check_consistency(). It should still be conditionally defined in the operations vector under #if MACH_ASSERT.

All the places where the two coherency routines are called need to be changed to call indirectly through the operations vector, and appropriate macros for this should be defined in svm_coherency.h.

The function m_get_writer() in norma/svm_state.c contains an assertion that will be wrong if more than one writer is allowed. This function is actually used for finding out whether a page is in use at all on any kernel as well as for getting the writer. The assertion should be removed and a similar check to enforce the single writer policy should be placed directly where m_get_writer() is called in strong_start_kernel_operation().

4. NUMA Support

The EMMI routines for pagein (memory_object_data_supply()) and pageout (memory_object_data_return()) pass out-of-line data. When the message crosses node boundaries, data is transmitted between nodes along with the message. This is obviously necessary on a NORMA system, but it is a waste of time on a NUMA architecture where the receiving node may access the data directly if it has a physical address (assuming, of course, that the memory access time is faster than sending out-of-line data in a Mach message). Normally the out-of-line data is internally represented by a vm_map_copy_t of one of two types: VM_MAP_COPY_OBJECT or VM_MAP_COPY_PAGE_LIST. By defining a new copy object type that contains physical addresses (or some sort of address meaningful to the particular machine), data can be passed between kernel and memory manager much more efficiently. This is somewhat outside the scope of supporting multiple coherency policies in XMM, but XMM contains some assumptions about copy object types that need to be generalized, and there are EMMI changes necessary to enable efficient use of the physical address passing.

4.1 New Copy Object Type

The new copy object type will be VM_MAP_COPY_PHYS_ADDR and it needs to be defined in vm/vm_map.h. In addition, a new member for this type needs to be added to the union in the vm_map_copy structure. In vm/vm_map.c, there needs to be a new flavor of vm_map_copyin() and vm_map_copyout() to handle the new type of copy object, and vm_map_copy_discard() needs to include a case for discarding the new type.

More specifically, the new VM_MAP_COPY_PHYS_ADDR type needs to be a list of physical addresses. Since these addresses represent pages that have already been allocated, there is no need to use the continuation logic used for the VM_MAP_COPY_PAGE_LIST type. A new member of the union in the vm_map_copy structure should be defined:

```
struct {                            /* PHYS_ADDR list */
        int npages;
        phys_addr_handle_t *addr_list;
} c_a;
```

The typedef of phys_addr_handle_t should be in a machine-dependent file, and addr_list will be a dynamically allocated list of physical addresses. The npages variable will be set to reflect the number of pages is the list of addresses.

The data argument in memory_object_data_supply() and memory_object_data_return() represents a page of out-of-line data. If memory_object_data_request() as received by the pager always has the address of the preallocated page (as should be the case), then the MIG definition of memory_object_data_supply() can be changed so that the data argument is the list of physical addresses. Unfortunately, memory_object_data_return() needs to be able to support both real out-of-line data and a physical address list. A data_return from the kernel may be initiated by the pageout daemon or it may result from a lock_request, which itself may come from either the pager or another kernel. If the data contained in the data_return message always went to the pager, then the MIG definition could be changed as for data_supply. However, the data is sometimes sent to other kernels, and the physical address list is not always appropriate. Therefore, a separate interface is needed to send the physical address list to the pager: memory_object_data_return_addr().

The choice of which interface to use will be made in m_user_data_return() in norma/xmm_user.c based on the type of the copy object.

The callers of memory_object_data_return(), namely vm_pageout_page() in vm/vm_pageout.c and memory_object_lock_request() in vm/memory_object.c, need to use the new vm_map_copyin_phys_addr() instead of vm_map_copyin_object().

The remaining work to be done is in XMM and NORMA, to allow for the new type of copy object. In norma/svm_request.c, the IS_GOOD_COPY macro needs to allow the the new type. In svm_satisfy_kernel_request(), the data is converted to page list format if it's not already in that format. This needs to be modified to allow a physical address list format to go through unchanged, since memory_object_data_supply() will accept either. In norma/svm_pagecopy.c, the argument to svm_copy_page() is a copy object that could be in the new physical page format, so svm_copy_page() needs to handle the new type.

The NORMA IPC code currently sends out-of-line data only in page list format. This is done by ipc_kmsg_copyout_to_network() in norma/ipc_output.c. Instead of always converting to a page list, it should also allow the physical address list to be used. Further modification in NORMA is necessary, but how this is done depends on how NORMA messages are transmitted and received on the particular architecture. The general idea is that the physical address list should be sent through NORMA as is.

4.2 Preallocating Page for Pagein

For pagein, in order to take maximum advantage of this technique, it is necessary for vm_fault_page() to preallocate a real page and then tell the memory manager where to put the data. Normally, a fictitious page is put into place until the data actually arrives, at which time the fictitious page is converted to a real page. In this case, the real page will be allocated before contacting the pager, and it needs to be converted back to a fictitious page in vm_fault_page() if the pager sends back memory_object_data_error(). In order to inform the memory manager the location of the preallocated page, a data argument needs to be added to memory_object_data_request(). This data argument can be set to NULL to use the conventional method, or it can be the physical address (list) for the new method.

In a system using XMM, there are two possibilities for how the page of data will be supplied in response to the request. The page might not exist on any kernel, in which case the pager is contacted for the data and it will place the data directly into the specified page. The other situation is that the page might have been instantiated on another kernel, and this time the other kernel will be contacted for the data. However, in this case the data already has a physical address which is the one that should be returned to the original requestor. At first glance it seems that it would be necessary to have a couple more types of copy object in order to distinguish between the data being put where specified or already existing somewhere else; but the only entity that cares is the original requestor, vm_fault_page(). Even though it would not be strictly necessary for the address to be passed back in memory_object_data_supply() when the pager has already placed the data into the page, if it returns the address anyway, the receiving side of memory_object_data_supply() can check the returned address against the one that was provided originally and deduce what happened. If it sees that the data is at a different location than the preallocated page, then it can convert the preallocated real page back to a fictitious page, free the real page, and proceed as would have ordinarily happened.

Until very recently, the code to preallocate a page existed in vm_fault_page() immediately following a comment about needing to reserve a real page in advance for the default pager. This is the place where the page should be preallocated for the new use of memory_object_data_return(). This preallocation may be done unconditionally unless there is a requirement that the preallocation be done only in certain instances. The following code should be put into vm_fault_page() in the block beginning with "if (look_for_page && !must_be_resident)" between checking pager_ready and checking for absent_count being greater than vm_object_absent_max:

```
if (m->fictitious && !vm_page_convert(m)) {
        VM_PAGE_FREE(m);
        vm_fault_cleanup(object, first_m);
        return(VM_FAULT_MEMORY_SHORTAGE);
}
```

Note that vm_page_convert() is the routine that converts a fictitious page into a real one. Then, the call to memory_object_data_request() needs to be changed to include the page's physical address as an argument.

Next, memory_object_data_supply() (vm/memory_object.c) needs to be changed to accomodate the preallocated page and handle the instance where the data already existed somewhere else. The general work that must be done for this purpose occurs in the code block that handles a busy and absent page (i.e., there is an outstanding request for this page). The page should be freed under two circumstances: it is fictitious, or it is a real page but its physical address does not match the physical address that data_supply has. In the second case, the page corresponding to the new physical address needs to be instantiated on this node at this point. In the case where the page was supplied directly into the preallocated page, the vm_page_insert() should be skipped.

In addition, memory_object_data_supply() needs needs to be structured so that it can handle either a VM_MAP_COPY_PAGE_LIST or VM_MAP_COPY_PHYS_ADDR type of vm_map_copy_t. Since both types are lists of pages, the basic structure of the code can remain as is with the modifications in the code that actually handles the list.

Preallocated pages complicate the logic that handles memory_object_data_unavailable(), which is in vm_fault_page(). The code that currently exists will operate correctly, but it will always allocate a new real page to zero fill. However, trying to use the preallocated page is not straightforward since a zero fill is not done until the whole shadow chain is walked, a busy page is left in the top object while the others look for the page, and the zero fill page is put into the top object. An initial attempt to use the preallocated page for the zero fill would be to check the status of the page in the top object. If it is a fictitious page then a real one needs to be allocated, but if not, then the real page should be used and the following code that frees the page already in the object needs to be avoided. The result is that there may be two real pages allocated while the shadow chain is being walked, but the page in the top object will be used for the zero fill.

4.3 Invalidation on Pageout

When a pageout daemon chooses a page to evict and needs to clean it, the VM subsystem first makes the page inaccessible to preclude further modification, then it sends the data to the memory manager via memory_object_data_return(). In the existing system, XMM only permits one kernel at a time to have the page mapped with write permission, and thus either the page is cleaned by this kernel with no possibility of modification by others, or several kernels have it mapped read-only but marked dirty and/or precious, and the page is already unmodifiable. Once this restriction is removed, some other mechanism is needed to ensure that a page will not be modified after it is sent to the pager. If this is not supported by hardware, the logical place to handle this is in XMM. While the need for this arose from the change in XMM policy, the actual mechanism itself should not be dependent on any particular policy but rather should be triggered by explicit request from the kernel. The reason for this is to avoid having to keep track of some notion of which kernel actually "owns" or controls the physical page.

This requires an extension to the memory_object_data_return() interface to add the extra "real_page" boolean. One way to handle this is to add an extra argument, or to group together the dirty, kernel_copy, and real_page booleans into one bitfield argument. In the long term, the bitfield seems like the most reasonable choice since future extensions would not require a change to the function call but merely extra bits that could potentially be ignored by pagers who didn't understand them. However, in the short term it would be easiest to group real_page in with just kernel_copy, since both the default pager and the vnode pager already ignore kernel_copy. This way, the calls to memory_object_data_return in VM can use the kernel_copy field to communicate with XMM for both purposes and the real_page boolean need not be sent to the pager at all. In fact, since the real_page parameter is intended only for communication between the kernel and XMM, it should not be sent to the pager.

Note that if real_page is TRUE, then dirty is implied to also be TRUE, and kernel_copy should be FALSE.

The definition of proxy_data_return() in norma/xmm_proxy.defs needs to have the type of kernel_copy changed to int. The bits need to be defined in mach/memory_object.h:

```
#define MEMORY_OBJECT_DATA_RETURN_NULL              0
#define MEMORY_OBJECT_DATA_RETURN_KERNEL_COPY       1
#define MEMORY_OBJECT_DATA_RETURN_REAL_PAGE         2
```

The callers of memory_object_data_return() in VM need to be updated to use the new form of kernel_copy, first to preserve the existing usage of the kernel_copy argument, then to take advantage of the new real_page functionality if desired. Current single node and NORMA systems will not use this new functionality. The calls to memory_object_data_return() are found in vm/vm_pageout.c and vm/memory_object.c.

The declarations of all the *_data_return() routines in XMM need to have the type of the kernel_copy argument changed from boolean_t to int. These are found in xmm_export.c, xmm_import.c, xmm_server.c, xmm_split.c, xmm_svm.c, and xmm_user.c, all in the norma directory. Also, M_DATA_RETURN() in xmm_invoke.c needs the same change. None of the callers of *_data_return() and M_DATA_RETURN() in XMM need modification.

The actual real_page functionality needs to be implemented in XMM, in svm_kernel_supply() (norma/svm_request.c). The kernel_copy argument to svm_kernel_supply() needs to be declared as an int, and possibly renamed to something like kernel_info. When the data_return() was initiated by the kernel's pageout daemon, there will be no current request for the page in SVM. When svm_kernel_supply() notices this, it calls svm_dispose_of_data_properly() which will then send the page to the pager. In the case where real_page is TRUE and there is no request, the invalidation must be done and the page then must be sent to the pager. This will involve setting up a request, sending lock_requests to flush the page (without cleaning) to the other kernels that have it, and then calling M_DATA_RETURN() to send the page to the pager when the lock_requests are all finished. Other requests for the page must be held off until after the page is sent off to the pager. It is possible for an outstanding request to exist when svm_kernel_supply() is called. In this case, either the requests should be deferred until the pageout is finished (restarting the requests is triggered by the data_write_completed() message), or the requests should be allowed to finish but new requests should be held off. This behavior very closely mirrors the actions taken when the pager does a lock_request. Pager requests have priority over kernel requests, and most of the actions described for this new case are already done for a pager-initiated lock_request. Care must be taken to make sure that the originating kernel does not get a lock_request.

Expanding on the above, svm_pager_request(), should be called from svm_kernel_supply() if real_page is TRUE, right after the "if (dirty)" block of code. An assertion should be made at this point that dirty is TRUE and kernel_copy is FALSE. Also, the data must be in physical address format since otherwise a copy has already been made, and this operation would make no sense. At this point, the page's usage will have been updated for the kernel that supplied the page, and since it no longer is keeping a copy, it will not be asked to flush the page later. svm_pager_request() will set up an appropriate request and call into the state machine for further processing, which will handle or defer any existing requests. Then svm_kernel_supply() should return. If real_data is not TRUE, then the usual processing will occur. Note that the assertion at the end of the "if (dirty)" block will fail for policies other than strong coherency and it should be removed or be made conditional. This new use of svm_pager_request() differs from a real pager request in that the data already exists. It needs to be stored somewhere other than in req->data so that it is not used to satisfy outstanding requests. This will also serve to differentiate between this invalidation request and a real pager request. Since the request may not exist before calling svm_pager_request(), the data should be passed in as a new argument, and the call will be:

```
svm_pager_request(mobj, offset, MEMORY_OBJECT_RETURN_NONE, TRUE,
                  VM_PROT_ALL, XMM_REPLY_NULL, data)
```

A new field, vm_map_copy_t return_data, should be added to the prequest structure to store the returned data until it is safe to send it to the pager. When svm_pager_request() creates the new prequest, it should place the data in the new field but otherwise proceed as usual. The next problem arises in svm_start_pager_request(), which is called from svm_pager_request(). It assumes that if all access is being revoked, then the data needs to be returned. Since the data is already in hand, the code near the beginning that checks for VM_PROT_ALL needs to also allow MEMORY_OBJECT_RETURN_NONE if the prequest contains return_data. At the end of svm_-start_pager_request(), svm_start_kernel_operation() is called. This routine is the policy-specific routine, and it is not really an appropriate place to handle the page invalidation. Instead, if the request contains return_data, a new function, svm_invalidate_kernels(req) should be called. The new function will iterate over all kernels, flushing all that have any usage of the page, similar to a loop that currently exists in svm_start_kernel_operation(). Before returning, the function needs to check req->kp_count. If it is zero, all the lock_requests have completed (or none were generated), and the request state should be set to Kernel_Completed.

When all the lock_requests have completed, the request state will be Kernel_Completed and svm_process_request() will be called. If the state is Kernel_Completed, and the request is a Pager_Request, then the state is set to All_Completed, and then svm_satisfy_pager_request() will be called to finally send the data to the pager. However, in this new case, the data is in the prequest, not in the request itself. The best time to place the data into the request is at the time the state is changed from Kernel_Completed to All_Completed (when the request type is found to be Pager_Request).

5. Acknowledgments

The general ideas in this design were the product of a joint effort by Terrell Magee and Roger Sunshine from Convex, and David Black, Steve Sears, and Michelle Dominijanni from OSF Research Institute.