A New Approach to Distributed Memory Management in the Mach Microkernel

Stephan Zeisset, Stefan Tritscher, and Martin Mairandres
Intel GmbH

A New Approach to Distributed Memory Management in the Mach Microkernel

Stephan Zeisset, Stefan Tritscher, Martin Mairandres
Intel Corporation

ABSTRACT

In this paper we describe a new approach towards extending Mach virtual memory semantics across the node boundaries of a multicomputer system. At its core, the Advanced Shared Virtual Memory (ASVM) system employs algorithms from the realm of shared virtual memory that were adapted and extended to create a system that is efficient and scalable and supports full VM semantics, paging between node memories and efficient execution of SVM applications. Our performance measurements demonstrate ASVM's superior efficiency and scalability compared to its predecessor, the Extended Memory Manager (XMM) that is part of the Mach kernel's NORMA distribution.

1. Introduction

Multicomputer systems generally run different copies of their operating system on each of their processing nodes.To ease both programming and system administration, the more advanced multicomputer operating systems, among them OSF1/AD TNC [2], support the concept of a single system image. This provides the user with the illusion of a single workstation environment, including single IP, process and file name spaces.

To accomplish this, OSF1/AD TNC relies heavily on two additions to its Mach microkernel that are also known as NORMA (No Remote Memory Access) extensions: NORMA-IPC for extending interprocess communication and XMM for extending virtual memory semantics across the multicomputer's node boundaries. NORMA-IPC is used by the operating system for all kinds of internode communication and synchronization, while XMM provides copy-on-access semantics for remote task creation and shared memory semantics for the memory mapped file system and user created shared memory segments.

Unfortunately, the original design of NORMA-IPC and XMM did not perform well on large scale parallel machines. The main deficiencies of NORMA-IPC were its broken flow control in many-to-one communication scenarios and its poor utilization of the available communication bandwidth. The main deficiencies of XMM were its non-scalable approach to handling shared memory and an inefficient communication protocol.

These problems prompted Intel's Scalable Systems Division, which based the operating system of it's Paragon multicomputer system on OSF1/AD TNC, to redesign the NORMA-IPC and XMM subsystems. In this paper we will concentrate on the XMM rewrite effort and its outcome, the Advanced Shared Virtual Memory (ASVM) system.

2. Background

2.1 The Paragon multicomputer system

The Paragon is a multiprocessor system with distributed memory. Each of its nodes contains two or three i860XP processors that share a local memory with a size of 16 to 128 MBytes. One of the processors usually acts as a dedicated message passing processor while the others execute user tasks. The nodes are interconnected by a two dimensional mesh of wormhole routed communication channels with a raw bandwidth of 200 MByte/s in each direction. Existing installations reach up to 1792 nodes.

Three properties of the Paragon multicomputer system had a major influence on the design of ASVM: The high bandwidth and low latency interconnect, the great number of nodes a system can contain and the fact that there is typically one node with an attached disk drive for 32 nodes that are used for computation.

2.2 The Mach Kernel's VM system

The Mach kernel's virtual memory (VM) system provides the abstraction of memory objects. These are user managed entities which are represented by a VM object on the kernel side and a memory object port on the user side. The available physical memory is used as a cache for memory object contents. User level pager tasks are responsible for providing the initial contents of a memory object and for preserving the object's data when it is evicted from the cache. The External Memory Management Interface (EMMI) protocol is used between the kernel and the pager tasks to exchange page contents and access rights (see FIGURE 1).

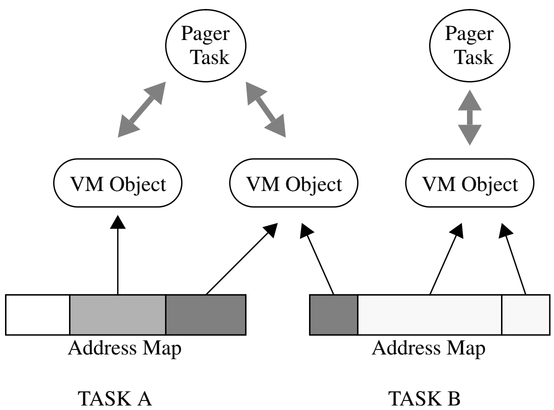

FIGURE 1 Mach VM System Structure

Beyond enabling a task to map a memory object into a specific part of its address space, the VM system also supports the shared access of multiple tasks to the same memory object (shared memory semantics) and the creation of lazy evaluated copies of a memory object (delayed copy semantics). While sharing of memory contents is easily accomplished by entering the same VM object into the address map of multiple tasks, delayed copy semantics are implemented by building shadow/copy relationships between VM objects.

Two different strategies are used for implementing delayed copy semantics, called symmetric and asymmetric copy strategy. With the symmetric copy strategy, the address maps of the source and the copy task continue to reference the same object. Only when a page is about to be written in either the source or the copy, a new object is created that has a shadow link to the original object. The new object replaces the original object in the address map where the write fault occurred (see FIGURE 2). While pages that are not present in the shadow object are retrieved from the source object through the shadow link, modifications of them take place in the shadow object. This means that the contents of the source object are frozen after a symmetric copy has been made.

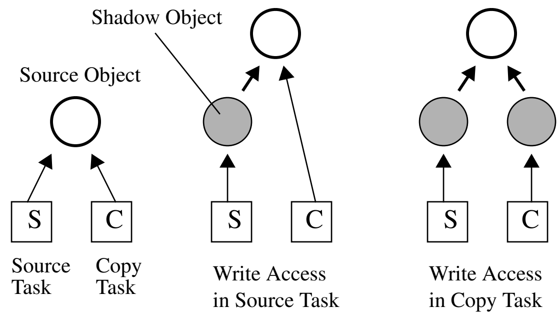

FIGURE 2 Symmetric Copy Strategy

While the symmetric copy strategy is a very efficient way of implementing delayed copy semantics, it is not applicable if changes in the source address space have to be reflected back to the source object's pager, such as with memory mapped files. Therefore, the asymmetric copy strategy is used in these cases. With this strategy, a copy object is created at the time a delayed copy operation is done. The source and the copy object are connected by copy and shadow links (see FIGURE 3). When a page is needed in the copy object, it is retrieved from the source object through the shadow link (pull operation). Before a page can be modified in the source object, first a copy of it is inserted into the copy object, using the copy link (push operation).

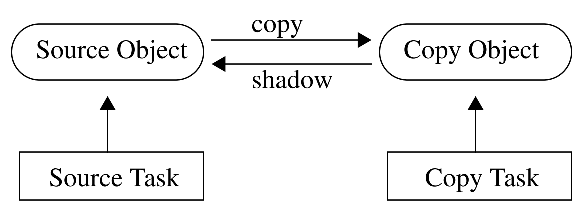

FIGURE 3 Asymmetric Copy Strategy

Multiple objects which are connected through copy links form a copy chain. New copies are inserted into the copy chain immediately after their source object.

An interesting property of both the symmetric and asymmetric copy strategies is that the creation of page copies is delayed until the page is modified in the source or copy address space. Pages that are retrieved through a shadow link in lieu of a read page-fault are not copied to the object where the fault occurred. Instead, the page-fault is satisfied by directly entering the source object page into the physical pagemap of the faulting task.

2.3 The eXtended Memory Manager

The original design of XMM is due to Joe Barrera, who created it in the context of a Ph.D. thesis at Carnegie Mellon University. Later, the Open Software Foundation (OSF) took over responsibility for further development of XMM [6]. In this section we describe the OSF NMK13 version of XMM, which is part of the software basis for the Paragon operating system and forms the basis for our performance comparisons. The major accomplishment in later OSF versions of XMM is a better integration of shared memory and delayed copy management, which closes a semantic gap in NMK13 XMM that prohibits combined use of shared and inherited memory.

2.3.1 Structure

XMM is located inside the Mach kernel and intercepts the communication between the VM system and external pagers. For each memory object XMM representations of the object are created on all nodes that make use of it. Only one of these representations holds state information and communicates with the pager task, while the others merely act as forwarding proxies for requests to and from their local VM system. This way XMM acts towards each node's VM system as the memory object's pager and towards the pager as a single node's VM system (see FIGURE 4).

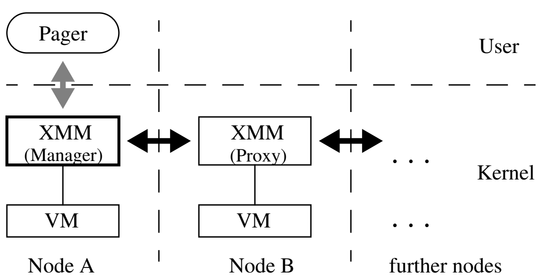

FIGURE 4 Global Structure of XMM

2.3.2 Shared Memory Management

XMM uses the centralized manager model for providing a coherent address space across multiple nodes. The manager node keeps track of the status of each page in the memory object's virtual address space on each node that makes use of the object and enforces a "single writer or multiple readers" policy. When some node makes a request for read or write access to a page of the memory object, this request is answered in two steps:

First, a coherent version of the page is created at the pager. This means that the page contents have to be returned to the pager if the page has been modified. Also, if the request is for write access to the page, the page has to be flushed from the VM cache of all nodes except the origin node of the request.

Then the request is forwarded to the pager, who can now view the requesting node as the only user of the page. The pager's answer is then forwarded to the requesting node.

2.3.3 Delayed Copy Management

NMK13 XMM supports delayed copy optimizations only in the context of remote task creation. The approach it takes toward the lazy evaluation of inherited memory is quite simple in that it leaves most of the work to the virtual memory system of the node where the source task is located.

This is done by creating a copy of the source address space just as in the case of a local fork() operation. Then a XMM internal pager is created for each memory object in the copy address space, thereby providing a new memory object abstraction to be mapped in the address space of the remote task. Page faults in the remote task's address space cause a request to the XMM internal pager which in turn generates a pagefault on the local copy address space and supplies the resulting page contents back to the remote node.

3. The ASVM system

3.1 Design Principles

Based on our experiences with XMM, we developed our new system along the following guidelines:

- **Distributed Manager**

  XMM is based on a centralized manager approach for managing cross-node memory relationships. This means that for each Mach memory object, the XMM code on a single node is responsible for coherency and copy management. Several research studies ([1], [10]) indicate that this approach is non-scalable and thus unsuitable for management of shared virtual memory on large systems, because the centralized manager becomes a bottleneck when a large number of nodes uses the memory object.

ASVM uses a distributed manager approach where each page has its own manager, which is also called the page owner. The ownership can migrate between all nodes that use the page.

- **Limited Memory Requirements**

  With XMM, the centralized manager stores the page state of a memory object in a data structure that requires 1 byte of non-pageable memory for each page in the virtual address space of the memory object, multiplied by the number of nodes that use the object. With large numbers of nodes or large and sparsely populated address spaces this concept can consume a lot of memory. In extreme cases it may even consume more memory than is actually available, leading to a system crash.

  ASVM not only distributes the page state information across the system, but also ties it to physical pages in that a node only holds state information about pages that are cached into its physical memory. The same concept is used by the Mach kernel's virtual memory system to support large and sparsely populated address spaces.

- **Asynchronous State Transitions**

  One problem of the mechanism XMM uses for implementing its delayed copy support is that the copy pager thread which generates a page-fault is blocked until the page-fault completes. As an internode copy chain might cross the same node multiple times, this leads to a deadlock if the available number of threads is exhausted.

  To avoid problems of this kind, ASVM generally uses asynchronous state transitions, which means that neither a thread nor other resources are blocked while waiting for a request to be answered.

- **Specialized Communication Protocol**

  XMM uses a protocol called XMMI (eXtended Memory Management Interface) for communication inside the XMM system. XMMI is an extension of the EMMI protocol that defines the interaction between the virtual memory system and an external pager [4]. Although this is an elegant solution, it is also very inefficient when used for coherency management, as it requires more messages than necessary. For example, transferring a write permission from one node to another using XMMI takes five messages, two of them containing page contents. With a more suitable protocol, this number could be reduced to three messages (request to manager, request from manager to current writer, and answer from current writer to new writer), only one of them containing page contents [7].

  ASVM uses XMMI only as an interface to the local VM system and pager while defining its own ASVM protocol for all communication between the ASVM instances on different nodes.

- **Dedicated Transport Service**

  XMM uses Mach NORMA IPC as a transport service for XMMI communication to remote nodes. This introduces high latencies and overhead due to the handling of port rights and complex message structures. This is especially conspicuous on multiprocessor systems with a high-performance interconnect such as the mesh used in Paragon systems. In fact, on these systems NORMA IPC is responsible for about 90 percent of the latency involved in resolving remote page faults for memory that is shared through XMM.

  The ASVM protocol, on the other hand, is mapped to a dedicated transport interface, the SVM transport service (STS). This allows the use of lower level protocol stacks for ASVM communication that can take advantage of the simple structure of ASVM messages, which consist of a fixed size block of untyped data (currently 32 Byte), possibly followed by the contents of a VM page (8 KByte). Also, flow-control is simplified by the fact that page contents are only transferred on behalf of a request from their receiver, which allows preallocation of page receive buffers.

3.2 Basic Operation

For managing coherency and memory inheritance, some entity must maintain information about the current state of a page. This entity is called the manager of the page. The main design freedom for any SVM system is the method used to distribute the managers of particular pages across the nodes of the system. There are several known approaches to this problem [1]:

- Centralized Manager

  The managers for all pages of a memory object are located on the same node. This approach is implemented in NMK13 XMM.

- Fixed Distributed Manager

  The page managers are statically distributed by using some function which maps a page number to a node number. The manager for the page is located on that node.

- Dynamic Distributed Manager

  The manager of a page migrates between the nodes that use the page. The node on which the manager of a page is currently located is called the owner of the page. Kai Li [1] suggests moving page ownership to the node that most recently made an request for the page and locating the actual page owner by following a *hint chain* across all the nodes that previously owned the page. This hint chain naturally collapses when a node that once owned a page becomes its owner again. The hint chain can also be collapsed while forwarding a request as the originator of the request becomes the next owner.

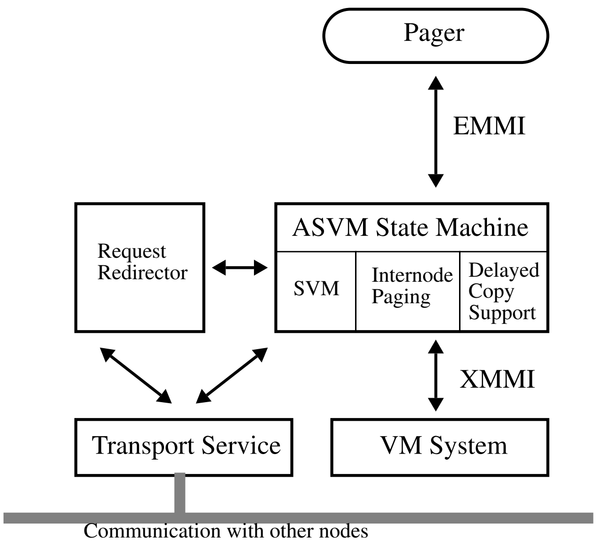

FIGURE 5 Structure of the ASVM System

ASVM adopts the dynamic distributed manager approach, but makes a clearer distinction between the method for distributing page managers and the method for finding the current manager of a page. In this approach, there are actually two managers for a page. One of them is the page owner, which keeps information about the state of the page and is responsible for answering access requests from other nodes. The other is the ownership manager, which is responsible for forwarding requests to the page owner. While a page is always owned by the node that most recently had write access to it, multiple methods are available for forwarding requests to the page owner. This is different from Kai Li's approach, in which page ownership also migrates on read requests and a single method is used for forwarding requests (the hint chain).

3.3 General Structure

FIGURE 5 shows the structure of the ASVM system and how it is embedded into the Mach kernel.

When the virtual memory system of a node needs access to some page, an access request is generated and fed into the request redirector, which is responsible for delivering it to the current page owner. When the request arrives on the page owner node, it is fed into a state machine which keeps the page coherent throughout shared usage, delayed copying, and internode paging.

3.4 Forwarding Mechanisms

The motivation for providing multiple strategies for finding the owner of a page comes from the design goal of avoiding data structures that grow with the size of virtual address spaces. This goal is guaranteed for the page management structures by enforcing the invariant that each node that is owner of a page has the page in its virtual memory cache. If no node has the page in its VM cache, there is no owner of the page and no state information about it. The pager can be viewed as the page owner in this case.

A different approach must be taken for page ownership information, since each node must also hold ownership information for pages that aren't in its virtual memory cache. Consider the following forwarding strategies:

- Dynamic forwarding

  Requests for a page are forwarded as in Kai Li's dynamic distributed manager approach.

- Static forwarding

  Requests for a page are forwarded through a fixed distributed ownership manager.

- Global forwarding

  Requests for a page are forwarded through all nodes until they eventually reach the page owner or - if there is no owner of the page - the pager.

Dynamic forwarding requires the most memory since each node has to hold an ownership hint for each page in the virtual address space. Static forwarding uses less memory since each node only holds ownership information for a subset of the pages in the virtual address space. Global forwarding uses the least memory since only the page owner itself has to know that it owns the page.

ASVM implements all three of these strategies and holds ownership information for dynamic and static forwarding in caches for the most recently accessed pages (see FIGURE 6). This means that both dynamic and static forwarding can fail if required information is not present in the caches. Static forwarding will not fail as often as dynamic forwarding since the static forwarding cache is in effect distributed among all static ownership managers and thus can hold many more owner references than a dynamic cache of same size. Therefore, static forwarding is used as a backup for dynamic forwarding and is in turn backed up by global forwarding. Global forwarding will never fail if any node is owner of the page.

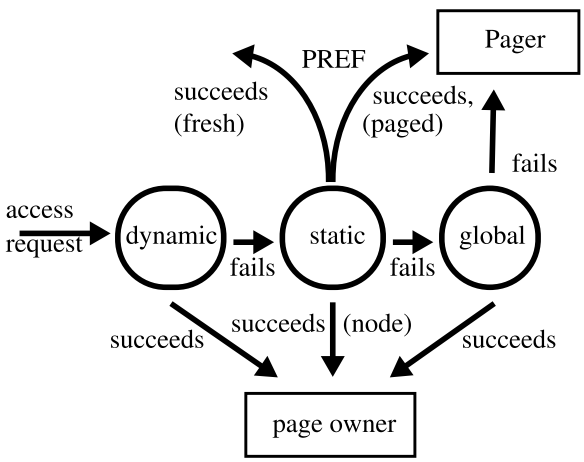

FIGURE 6 Forwarding Mechanisms

In addition to a node reference the static cache can also hold hints that the page has never been initialized (fresh) and that the page has been paged out (paged), thereby avoiding costly global forwarding operations in these cases.

The ASVM system allows to disable either dynamic or static forwarding (or both) on a memory-object basis. This provides great flexibility. If only static and global forwarding are enabled, the behavior of the ASVM system is identical to Kai Li's fixed distributed manager approach. Enabling dynamic forwarding makes the ASVM system resemble the dynamic manager approach. Of course this assumes that in both cases the relevant caches are big enough.

3.5 Shared Memory Management

The only coherency model that is currently supported by ASVM is *strong coherence*, which means that any read operation to a shared memory address will return the data of the most recent write operation to this address.

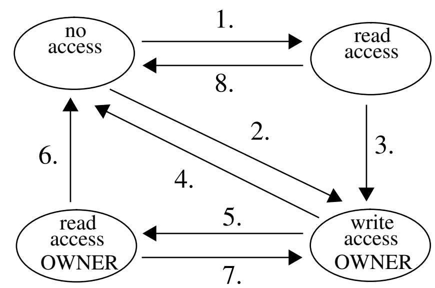

FIGURE 7 SVM state diagram

FIGURE 7 shows the part of the ASVM state machine that keeps a page coherent through shared usage, enforcing the single writer or multiple readers invariant. The state of a page on a particular node includes the type of access the node's VM system has to the page and an indicator if the node is owner of the page. The following state transitions can occur:

1. The node is granted read access to the page.
2. The node is granted write access to the page.
3. The node is granted an upgrade from read to write access.
4. The node is owner of the page and grants write access to another node.
5. The node is owner of the page and grants read access to another node. This node is also entered into a list of nodes with read access (reader list)
6. The node is owner of the page and grants write access to another node. An invalidation message is sent to all nodes in the reader list.
7. The node is owner of the page and upgrades its own access rights from read to write access. An invalidation message is sent to all nodes in the reader list.
8. The node receives an invalidation message from the page owner.

3.6 Internode Paging

The main idea behind the concept of internode paging is to extend not only the semantics of the VM system across node boundaries, but also its concept of managing physical memory as a cache for virtual memory contents. Together with the mechanisms for maintaining coherency, ASVM's internode paging facilities allow it to view the physical memory of all nodes that use a particular memory object as a cache for the memory object's contents.

When a page is evicted from a node's virtual memory cache, the following algorithm is used:

1. If the node is not the page owner, the page is simply discarded, as it can be retrieved from the page owner at any time.
2. If the node is the page owner and its list of nodes with read access is not empty, these nodes are asked, one after another, if they still have read access to the page (they might have discarded the page as in Step 1). If a queried node answers to have no read access to the page, the node is removed from the reader list. Else ownership for the page is transferred to this node. Note that this ownership transfer doesn't require sending the page contents.
3. If the node is the page owner and there are no more nodes with read access to the page, ASVM tries to transfer the page to one of the other nodes that have mapped the memory object to which the page belongs. This fails if no node with sufficient free memory can be found.
4. Finally, if Step 3 failed the page is returned to the memory object's pager.

For selecting a pageout node in step 3, a counter is used that cycles through the list of nodes which have mapped the memory object. This counter is incremented on each pageout. First, the node identified by the current counter value is asked to accept the page transfer. If this node doesn't accept the page transfer because it is low on memory, the node which most recently accepted a page transfer is asked again.

This algorithm is meant to adapt itself to the current memory situation by locking onto nodes which are known to have free memory available until another node is found which also has memory available. If many nodes have free memory, such as when a single node initializes a SVM region that is mapped on multiple nodes, the pages are distributed evenly among the other nodes. This is provides good load balancing for ASVM's distributed manager algorithms.

3.7 Delayed Copy Management

ASVM extends the asymmetric copy strategy of the virtual memory system across node boundaries. The basis for internode copy relationships is provided by the shared virtual memory functionality of ASVM. When a copy of a memory object is to be mapped on some node, first a shared mapping of the source object is established on this node. Then a local copy is created through the standard mechanisms of the VM system (see FIGURE 8). After that all resident pages of the source object on all nodes which share the object are marked read only by sending a message to the sharing nodes which causes them to do a memory_object_lock_request on the copied region.

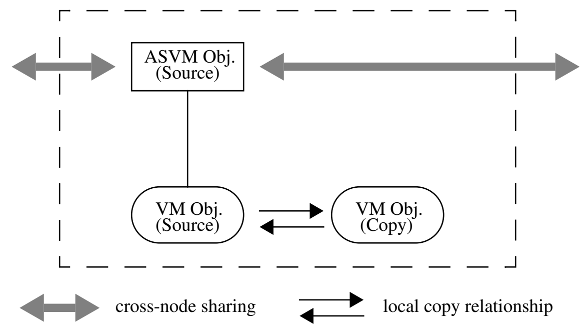

FIGURE 8 Delayed Copy Layout

While all local push and pull operations continue to be done internally in the VM system, ASVM has to be involved if either the source or the copy object are shared between multiple nodes:

- If the source object is shared, a page which is about to be modified has not only to be pushed to the local copy object, but also to copy objects on all other nodes which share the source object.
- If the copy object is shared, before pushing a page to the copy object it has to be determined if the page is already present on any of the nodes which share the copy object.

3.7.1 Extensions to the EMMI Interface

The shared virtual memory functionality of ASVM was implemented without changing the EMMI. This was not possible with the delayed copy management, because the EMMI design hides the existence of copies and shadows from the memory manager. So the following EMMI extensions had to be made to give the memory manager control over the VM internal copy mechanisms:

memory_object_lock_request was extended by a "mode" argument that allows to specify if the page should be pushed down the VM internal copy chain before executing the lock operation.

memory_object_lock_completed was extended by a "result" argument which is used to return an indication if the lock_request couldn't execute a push operation because the page was not present in the VM cache.

memory_object_data_supply was extended by a "mode" argument that allows to push a page down the copy chain instead of supplying it to the source object.

memory_object_pull_request was added to allow a page to be retrieved through the VM internal shadow chain.

memory_object_pull_completed was added to return the result of a pull_request. There are three possible results:

1. The page is not available and can be zero-filled.
2. The page is available and its contents are returned.
3. The memory manager of a shadow object has to be asked for the page and the shadow object port is returned.

3.7.2 Push operations

A page has to be pushed into all (possibly remote) copies of a memory object before it can be modified in the source object. A push operation is initiated by the current owner of a page if he receives a write request and the page has not already been pushed into all copy objects.

To determine if a page needs a push operation, ASVM uses version counters for memory objects and pages: An object's version counter is incremented each time a copy is made from the object. The version counter of a page is set to the associated object's version counter each time a push operation takes place on the page. If a page is about to be modified and the page version is not equal to the associated object's version, a push operation is initiated before write access is granted.

When a push operation takes place for a page, a message is sent to all nodes who have mapped the associated memory object, except the node who initiates the push operation. On these nodes the memory_object _lock_request EMMI call is used for prompting the VM system to push the page down the copy chain and invalidate it in the source object. If the page is not present in the VM cache, the reply to the lock request will indicate this and a request to send the page contents is sent to the page owner. Once the request is answered, a memory_object_data_supply EMMI call is made to push the page down the copy chain.

Once the page owner received replies from all nodes which were requested to push the page and after all missing pages have been sent to these nodes, the source object side of the push operation is completed. Additional effort is required on the side of the copy objects if they are shared. To determine, if a page is already present in a shared copy object, a special type of access request, called a push scan request, is generated and forwarded through ASVM's forwarding mechanisms. If a page owner exists in the copy object, it will answer the push scan request and the push operation will be canceled for this copy object. If no page owner exists in the copy object, the request will finally be forwarded to the shadow object and the push operation will proceed for this copy object.

3.7.3 Pull operations

For pull operations, ASVM uses the VM system to traverse local shadow chains and its own SVM capabilities to bridge the gap between node boundaries. If a page-fault occurs in a VM object, the VM system will traverse the local shadow chain and - if the page isn't present in one of the traversed source objects - generate a memory_object_data_request EMMI call in the first object that has an associated ASVM object. ASVM will then forward this request through its forwarding mechanisms. If the page is present on some node, it will have an owner who receives and answers the request.

Else - assuming that the current object is a copy too - the request will finally be forwarded to the node on which the copy was created and that also has a mapping of the corresponding source object. On this node, which is also called the peer node of the current object, ASVM will then use the memory_object_ pull_request EMMI call to traverse the local shadow chain. If the requested page is present in one of the source objects on this node or can be zero-filled, it will be supplied to the origin node of the request.

If the result of the pull_request indicates that the page might be present in a source object which has an associated ASVM object, the request will be forwarded into this object. This continues until either the page is found in some object or the end of the shadow chain is reached, in which case the page can be zero-filled.

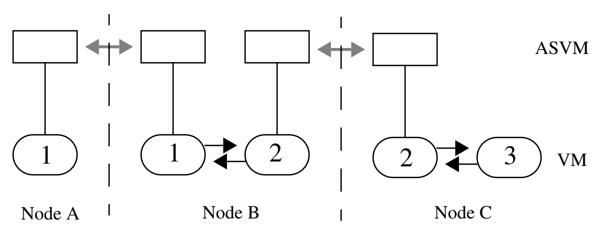

FIGURE 9 Copy chain across multiple nodes

For an example, FIGURE 9 shows a copy chain across two nodes as it is created if a task forks to a remote node and the child task does the same. Assume that a page-fault occurs in object 3 on Node C and the page is located in object 1 on Node A. The VM system on

Node C issues a data_request for the page in object 2. ASVM forwards the request to Node B, which is the peer node of object 2, and uses a pull_request to traverse the local shadow chain. The result of the pull_request indicates that the page has to be looked up in object 1 and ASVM forwards the request to Node A. Here, again a pull_request is used and returns the page contents. ASVM then supplies the page to the object from which it got the request, object 2 on Node C. Finally, the VM system will take care of entering the page into the physical pagemap of the task that took the page-fault in object 3.

One issue left out so far is that of synchronization between push and pull operations. Because of the distributed nature of ASVM's page management, a request from a copy object can enter its source object at the same time a push operation is in progress. In this case, the copy request is held up until the push operation completes. Then a retry indicator is set in the request and it is sent back to the origin node, which causes the request to be repeated.

4. Performance Measurements

We made our performance measurements on a Paragon multicomputer system with 72 GP nodes (2 processors and 16 MByte memory per node).

4.1 Basic Page-Fault latencies

4.1.1 Page-Faults on Shared Memory

Table 1 compares characteristic types of SVM page faults and their latency under the ASVM system and NMK13 XMM. All latencies were measured in user-task context by performing read or write operations and timing their duration (in milliseconds).

FIGURE 10 is a graphical representation of the latency introduced by a write page fault in relation to the number of nodes that have read copies of the page. The figure distinguishes between two cases. In one, called *write upgrade fault*, the faulting node already has a read copy of the page. In the other, called *write fault*, the faulting node doesn't have a read copy. The NMK13 XMM times are measured for the general case in which the XMM stack is remote from both the faulting node and the nodes that have read copies.

The graph shows that the page fault latencies of ASVM increase much slower with the number of nodes that have a read copy of the page than the page

Table 1: Page Fault Latencies

Fault Type	ASVM	XMM
Write fault on a page with 1 read copy	2.24	38.42
Write fault on a page with 2 read copies	3.10	12.92
Write fault on a page with 64 read copies	8.96	72.18
Write fault on a page with 2 read copies, faulting node has read copy	1.51	3.83
Write fault on a page with 64 read copies, faulting node has read copy	7.75	63.72
Read fault on a page, faulting node is first reader	2.35	38.59
Read fault on a page, faulting node is second reader	2.35	10.06

fault latencies of NMK13 XMM. This is especially important for SVM applications and is one indicator for the better scalability of ASVM.

The huge difference between page faults with only one read copy and with two read copies that was measured for NMK13 XMM can be explained by the fact that XMM writes a dirty page to the paging space when it is requested for the first time by another node.

4.1.2 Page-Faults on Inherited Memory

For evaluating the performance of delayed copy operations across node boundaries we used a test program that initializes a region of memory (128 KByte), spawns a chain of copies of that region across a defined number of nodes and faults in all pages of the region on the last node in the copy chain. FIGURE 11 shows the resulting page fault latencies both for NMK-13 XMM and ASVM. They can be described as $lb + n * la$, where lb is the basic latency for a remote copy-on-access fault (5.0 ms for NMK-13 XMM and 2.7 ms for ASVM) and la is the cost for forwarding the page fault across an additional node (about 4.3 ms for NMK-13 XMM and about 0.48 ms for ASVM).

The low additional costs ASVM incurs on faults that have to be forwarded across a long copy chain is espe-

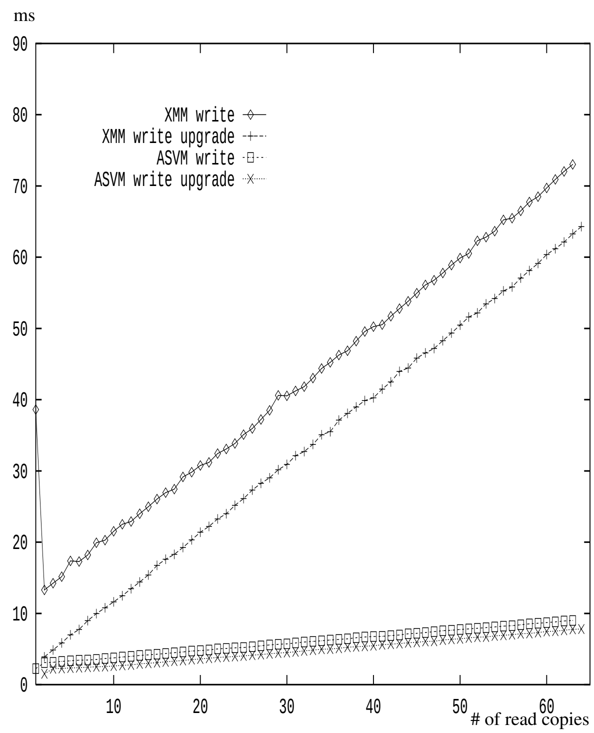

FIGURE 10 Write Fault Latencies for a Page with N Read Copies

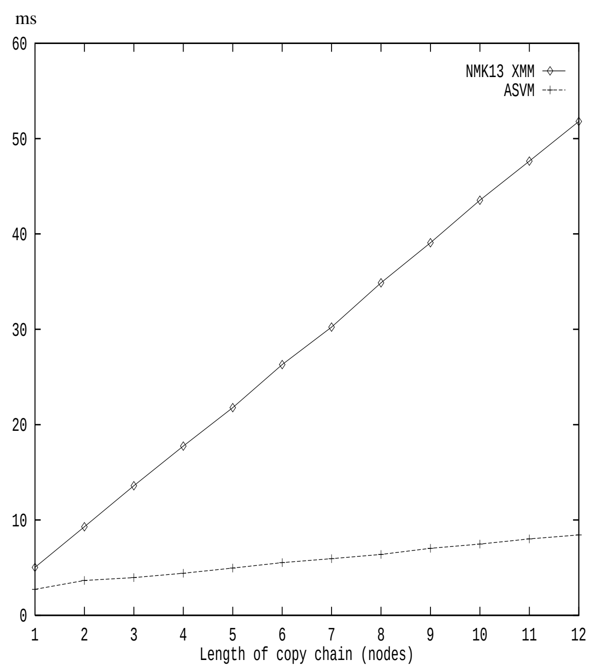

FIGURE 11 Latency for page faults across a copy chain

cially important for applications that use dynamic load balancing, as each migration of a task adds another stage to the copy chain from the node where the task has originally been started to the node where it is running. But also normal parallel applications take a considerable advantage: The Paragon operating system spawns applications to a set of nodes by forking along a binary tree, which creates copy chains of a maximum length that grows logarithmically with the number of nodes involved. For example an application which is started on 256 nodes creates copy chains with a maximum length of 8. A page fault across a copy chain of length 8 is associated with a latency of 35 ms for NMK-13 XMM and 6.4 ms for ASVM.

4.2 Mapped Filesystem Performance

While the Paragon OS uses a memory mapped unix file system, parallel access to the same file is sequentialized by the OSF1/AD TNC server, which lacks multiple reader semantics. Therefore, the measurements in this section don't use standard Unix read/write calls. Instead, they bypass the server by using the mmap() function to map a file into memory and reading/writing directly from/to memory. The performance of standard read/write calls can be expected to come close to the values presented here once multiple reader semantics are added to the OSF1/AD server. TABLE 2, FIGURE 13 and FIGURE 12 show the effective transfer rates seen by each of multiple nodes accessing the same file.

The write transfer rate is measured by letting all nodes write different sections of a 4-MB file. Asynchronous writes were used, so the upper limit for the combined transfer rate of all nodes is given by the rate at which the file pager can supply initially zero-filled pages for the file. The read transfer rate is measured by letting all nodes read a 4-MB file in parallel. The upper limit for the individual transfer rate of each node is given by the rate at which the file pager can supply the contents of the file.

TABLE 2 File Transfer Rates (MB/s)

	1	2	4	8	16	32	64
ASVM write	2.80	2.60	2.05	1.22	0.62	0.30	0.15
XMM write	2.15	1.77	0.90	0.49	0.24	0.12	0.06
ASVM read	1.57	1.53	1.14	0.91	0.70	0.66	0.66
XMM read	1.18	0.38	0.25	0.11	0.05	0.02	0.01

Note that because of its distributed manager concept, ASVM is able to sustain a reasonable read transfer rate even on a high number of nodes. To achieve the same

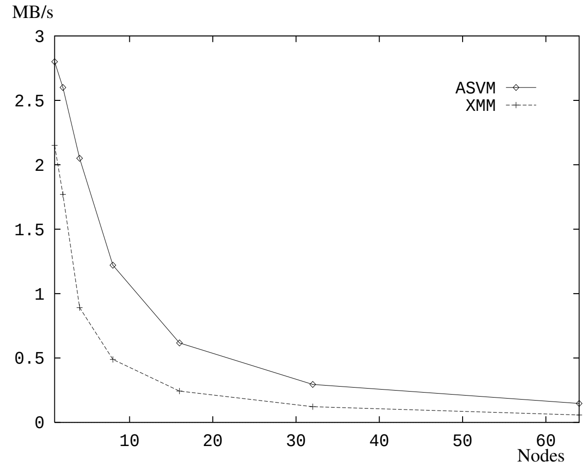

FIGURE 12 Parallel Write Bandwidth for Memory Mapped Files

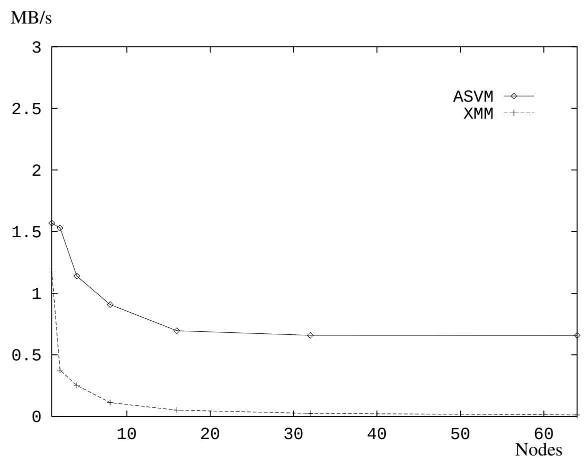

FIGURE 13 Parallel Read Bandwidth for Memory Mapped Files

for write transfers, the file pager would have to tell the kernel about chunks of uninitialized data, so that ASVM's distributed mechanisms could be used for supplying initially zero-filled pages.

4.3 SVM Application Performance

This section presents performance results based on a program called EM3D. This program's communication was originally based on active messages [9], which we changed towards shared memory based communication. EM3D models the three dimensional propagation of electromagnetic waves. Its basic data structure is a bipartite graph that has directed edges from a set of E nodes representing the electric field to a set of H nodes representing the magnetic field, and vice-versa. After an initialization phase for building up the graph, the computation consists of a large number of iterations to calculate the development of the fields over time. In each iteration the value of each E node is changed to the weighted sum of the H nodes to which it is connected, and then the value of each H node is changed to the weighted sum of the E nodes to which it is connected. To prevent confusion, the E and H nodes of the EM3D code will be called cells.

The graph for the performance measurements was generated randomly with a user-specified percentage (20%) of the edges (6 per cell) leading to a cell located on a different processing node. The execution times are given in seconds for 100 iterations of the computation loop. The initialization phase wasn't included in the measurements since practical applications would use even more iterations, making the initialization overhead almost vanish.

TABLE 3 demonstrates the effect of ASVM's enhanced scalability on SVM applications and shows the execution times of EM3D for various problem sizes both for NMK13 XMM and ASVM. The column headings indicate the number of nodes on which the application is run while the row headings indicate the use of XMM or ASVM and the problem size, given by the number of cells. With ASVM the execution times decrease with the number of nodes, resulting in reasonable speedups while with NMK13 XMM the execution times actually increase, resulting in a slowdown.

TABLE 3 EM3D Timings[a] (seconds)

EM3D	1	2	4	8	16	32	64
ASVM 64000	43.6	32.0	19.9	13.9	11.2	9.86	9.55
XMM 64000	43.6	151	213	392	755	1405	2735
ASVM 256000	174*	**	**	33.6	21.5	15.6	12.8
XMM 256000	174*	**	**	520	842	1604	2957
ASVM 1024000	698*	**	**	**	**	54.2	24.4
XMM 1024000	698*	**	**	**	**	1863	3373

a. * estimated timings without paging.
   ** meaningless because of paging effects.

Note that each cell in the problem space needs 224 bytes of memory. This means that 64,000 cells consume 14 MB, which is already too much for a 16-MB processing node that only has about 9 MB of memory available for user applications. The sequential measurement with 64,000 cells was therefore made on a 32-MB node. The parallel measurements were omitted where the combined memory of the 16-MB compute nodes wasn't sufficient to hold the complete data set.

5. Conclusions

Our first evaluations indicate that ASVM fully achieves its goal of providing efficient and scalable distributed memory management. The main reason for this success lies in the close interaction between the three ASVM subsystems for shared virtual memory management, delayed copy support and internode paging. To demonstrate this interaction, we will take a closer look at how it helps in achieving scalability and efficiency:

- Scalability

  As described in the section 3.5, the owner of a page keeps a list of nodes with read access to the page. The maximum size of this list grows linear with the number of nodes in the system, which poses a problem for scalability. This is where ASVM's internode pageout mechanisms enter the game: The greater the number of nodes with read access to a page, the greater the number of nodes which are available for an ownership transfer when memory gets scarce and the page is evicted from the VM cache. Such, ASVM's internode pageout mechanisms effectively balance the amount of owner information each node has to keep among all nodes which share a memory region.

- Efficient Memory Usage

  A great part of the memory inherited to a child process is not modified by either the parent or the child. The most prominent example for this are program text segments. As ASVM uses the VM system for establishing local copy relationships, pages requested through a read page fault will be supplied to the source object on the requesting node, instead of the copy object (see 2.2). This means that paging for read-only pages of copy objects is done through their source object, taking full advantage of ASVM's internode pageout mechanisms. If, for example, a parallel application is loaded onto a number of nodes and memory gets scarce, ownership for all read-only pages will be distributed among the participating nodes and not owned pages are discarded. If a node later needs access to a page it has discarded, the page can be retrieved from its current owner instead of reading it from a disk.

6. Future Work

While ASVM provides a solid foundation for establishing an efficient and scalable single system image, a number of modifications has to be made to the rest of the Paragon OS to utilize ASVM's full potential. We will concentrate on the area of file systems here.

Currently, the Paragon OS provides two types of local file systems, the Unix File System (UFS) and the Parallel File System (PFS). The Unix File System is a memory mapped filesystem and thereby uses the VM system to provide data buffering directly on the nodes which use a file. On the other hand, the Parallel File System supports striping of files across multiple I/O nodes and uses NORMA-IPC to distribute read/write requests to the I/O nodes. The main advantage of UFS is its caching scheme that becomes even more efficient through ASVM's internode paging facilities, while the main advantage of PFS is its scalability through the use of multiple I/O nodes.

In the following we will hint at what is necessary to combine the advantages of UFS and PFS into a new filesystem that supports striping, local caching and full Unix file semantics without a loss of performance:

- Provide and utilize ASVM primitives for locking a range of pages in a shared address space for the exclusive access of a particular task on a particular node. This would allow to guarantee the atomicity of read and write operations by locking the range which is about to be modified prior to write operations. The current scheme uses NORMA-IPC to acquire an exclusive token from a token server each time a read or write operation takes place and the token is not present on the node which does the file access.
- Modify the VM system to allow multiple pagers for one VM object that are used for paging requests in a round-robin fashion. This allows one VM object to represent a striped file system, with one pager located on each of the I/O nodes.

- A clustering of page-out and page-in requests has to be implemented in the virtual memory system and in ASVM to achieve adequate bandwidths.

Acknowledgments

This research was sponsored by the Advanced Research Projects Agency under contract MDA972-89-C-0034. It has also benefitted from discussions, suggestions and feedback by many people, including our colleagues at Intel's Scalable Systems Division, Fritz Gerneth - Intel GmbH and Michael Gerndt - KFA Juelich.

References

[1] Kai Li: **Shared Virtual Memory on Loosely Coupled Multiprocessors**. Ph.D. thesis, Yale University, September 1986

[2] Roman Zajcew, Paul Roy, David Black et al: **An OSF/1 UNIX for Massively Parallel Multicomputers**. In Proceedings of the USENIX conference, January 1993.

[3] Richard F. Rashid: **Threads of a new system**. Unix Review Vol 4(8), August 1986

[4] Michael Wayne Young: **Exporting a User Interface to Memory Management from a Communication-Oriented Operating System**. Ph.D. thesis, Carnegie Mellon University, November 1989

[5] Avadis Tevanian, Jr.: **Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments**. Ph.D. thesis, Carnegie Mellon University, December 1987

[6] Bill Bryant, Steve Sears, David Black, Alan Langerman: **An Introduction to Mach 3.0's XMM Subsystem**. OSF Research Institute Operating Systems Collected Papers Vol. 2, October 1993

[7] Alessandro Forin, Joseph Barrera, Michael Young, Richard Rashid: **Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach**. Report CMU-CS-88-165, Carnegie Mellon University, August 1988

[8] Pete Keleher, Sandhya Dwarkadas, Alan L. Cox, Willy Zwaenepoel: **Treadmarks: Distributed shared memory on standard workstations and operating systems**. In Proceedings of the USENIX conference, January 1994

[9] Culler, D.E, Dusseau, A., Goldstein, S.C., et al.: **Parallel Programming in Split-C**. In Proceedings of the Supercomputing 93, 1993

[10] Lahjomri, Z., Priol, T.: Koan: **A Shared Virtual Memory for the iPSC/2 Hypercube**. In Proceedings of the CONPAR conference, 1992

The Authors

Stephan Zeisset currently works as an International Project Leader at Intel's Scalable Systems Division. He graduated in 1994 with a Masters degree in Computer Science from the Munich University of Technology. His masterthesis built the basis of the XMM rewrite project, in which he took part as a Software Engineer at Intel GmbH. His e-mail address is sz@ssd.intel.com.

Stefan Tritscher is a Technical Marketing Engineer at Intel GmbH. Prior to joining the marketing department he worked as a Senior Software Engineer on Paragon OS components such as load balancing and the XMM rewrite project. He is holding a Masters degree in Computer Science from the Munich University of Technology. His e-mail address is stefan@esdc.intel.com.

Martin Mairandres is currently pursuing a Ph.D. degree from RWTH Aachen, Germany. He is also holding a Masters degree in Computer Science from the Munich University of Technology. During his Ph.D. research at Intel GmbH he contributed a lot to the shared virtual memory functionality of ASVM and designed interfaces for system and application level monitoring. His e-mail address is martinX@esdc.intel.com.

Trademarks