Global Memory Management for a Multi Computer System

Dejan Milojicic, Steve Hoyle, Alan Messer, Albert Munoz, Lance Russell, Tom Wylegala, Vivekanand Vellanki,[†] and Stephen Childs[‡]

HP Labs, Georgia Tech,[†] and Cambridge University[‡]

[dejan, hoyle, messer, bmunoz, lrussell, wylegal]@hpl.hp.com vivek@cc.gatech.edu[†] Stephen.Childs@cl.cam.ac.uk[‡]

Abstract

In this paper, we discuss the design and implementation of fault-aware Global Memory Management (GMM) for a multi-kernel architecture. Scalability of today's systems is limited by SMP hardware, as well as by the underlying commodity operating systems (OS), such as Microsoft Windows or Linux. High availability is limited by insufficiently robust software and by hardware failures. Improving scalability and high availability are the main motivations for a multikernel architecture, and GMM plays a key role in achieving this. In our design, we extend the underlying OS with GMM supported by a set of software failure recovery modules in the form of device drivers. While the underlying OS manages the virtual address space and the local physical address space, the GMM module manages the global physical address space. We describe the GMM design, prototype implementation, and the use of GMM.

1 Introduction

GMM manages global memory in a Multi Computer System (MCS) by allowing portions of memory to be mapped into the virtual address spaces managed by each local OS. An MCS allows booting and running multiple operating systems on a single hardware architecture (see Figure 1) with cache coherent memory sharing among the nodes. Each node contributes its own physical memory divided in two parts. One is visible locally while the remainder contributes to the global memory, visible to all nodes (see Figure 2). The primary GMM benefits on a multi computer system are improved scalability and high availability. Scalability is improved beyond the scalability limits of a single OS, by allowing applications to run on any OS instance and some of them on multiple instances at a time, concurrently, while sharing memory and other global resources. Whereas the former require no modification, the latter require some amount of parallelizing and use of GMM and MCS interfaces.

Availability is improved compared to a large SMP by allowing other instances of local OSes, as well as applications on top of them, to continue running even if a single instance of an OS fails due to a software failure. Multi computer systems are especially well suited for enterprise data centers where applications, such as Oracle or SAP, require increased scalability and high availability.

GMM offers other benefits: first, the ability to use the fastest form of interconnect in an MCS system; second, the possibility of easy and fast sharing between nodes, following an SMP programming model; third, it allows for better resource utilization by allowing overloaded nodes to borrow memory from underutilized nodes; finally, it allows scaling of applications requiring memory beyond a single node (e.g. OLTP and data base).

GMM design goals consist of the following:

- scalability and high availability,
- shared memory within and among different nodes,
- a suitable environment for legacy applications designed to use shared or distributed memory,
- sophisticated support for new applications, and

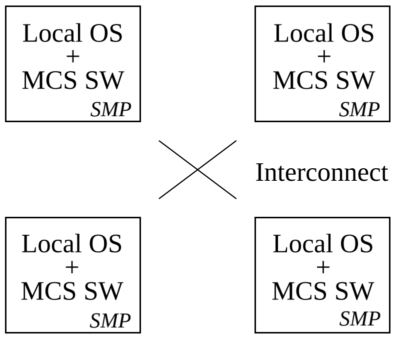

Figure 1 Multi Computer System (MCS) Architecture. *Each node (an SMP) runs a copy of the OS. Interconnect maintains cache coherent shared memory among nodes*

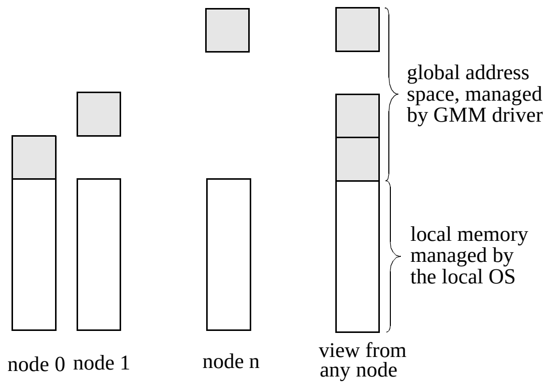
global address space, managed by GMM driver
local memory managed by the local OS
node 0
node 1
node n
view from any node

Figure 2 MCS Physical Address Space Organization. *Each node contributes a portion of its memory to global pool managed by GMM. Local memory is managed by the local OS.*

- good performance and resource sharing across nodes.

The rest of this paper is organized in the following manner. Section 2 presents GMM problems and non-problems. Section 3 presents the GMM design and Section 4 the prototype implementation. Section 5 overviews the use of GMM. In Section 6, we describe experiments. Section 7 presents the lessons learned. Section 8 overviews the related work. Section 9 concludes the paper and outlines the future work.

2 GMM Problems and Non-Problems

In the course of the GMM design we have identified problems that we considered important to address:

- **Recoverability.** Software failures on any node (e.g operating system failures) should not cause failure or rebooting of the whole system. GMM needs to recover its data from the failed node if it is still accessible. The memory occupied by a failed node needs to be freed up and inconsistent state needs to be made consistent (e.g. if a thread on a failed node died in the middle of updating data structures on another node). In order to be able to recover from the failure, the applications need to adhere to the recommended recoverable programming model (e.g. register for node crash events, replicate, checkpoint, etc.).
- **Memory scalability.** GMM is required to support access to more memory than is supported by a single OS, since there are *n* nodes each contributing to the physical memory of the whole MCS system. Therefore, limitation of 4GB of virtual address space size for ia32 is not acceptable [15, 16].
- **Local and remote sharing.** GMM must support memory sharing between threads on same and on different nodes, at both user (applications) and kernel (between MCS system components only) levels. This may require changes/extensions to the local OS APIs.
- **Usability and deployment.** The GMM recoverable programming model should not require significant changes to the existing applications. This is not such a strong requirement for MCS system components.
- **Minimal (if any) changes to the underlying operating system.** We used only extensions in the form of device drivers for prototype implementation. We also identified minimal changes to OS required for more sophisticated support (see Section 4.4 for details).
- **Globalization of resources and security.** Globally shared memory needs to be accessible for use from any node and it needs to be protected from misuse.

Based on experience from past systems and by adhering to Lampson's principles [18], we identified these problems that we decided to avoid solving:

- **Software distributed shared memory (DSM)**. In MCS, consistency is supported by hardware. Other DSM systems supported recovery as a part of their consistency model (e.g. [5, 17]). In the case of GMM, recoverability is considered as a separate issue.
- **Local to remote memory latency ratio**. Our assumption is that the remote to local ratio will be 2 or 3 to 1 and as such it does not justify implications on the design. Early NUMA architectures had over 10-15 to 1 ratio and they paid a lot of attention to data locality. However, GMM recoverability still imposes some location-awareness, e.g. for replication purposes.
- **There is no single system image aspect.** GMM does not strive for single system image support, such as in the case of Locus [25] or OSF/1 AD [31], or for a transparent extension of the local interfaces, such as in the case of Mach [2]. It is acceptable to use GMM by writing according to specific GMM interfaces.
- **Transparent fault tolerance is not a goal.** GMM should be recoverable, but it is acceptable that certain users of GMM fail if they do not adhere to recoverable programming model. GMM guarantees to a recover from a single node OS failure (blue screen) which is the most common failure on NT. Gray claims that most failures are due to software [14]. In this phase, we have not addressed hardware faults.

3 Design

3.1 MCS Overview

GMM is designed to use the MCS system and recovery components. The recovery components implement a recovery framework to keep track of the current state of the nodes in the system. In the presence of failures, they detect and signal the faults to MCS components. The MCS system components provide support for global locking and fast communication (see Figure 3). Inter-Node Communication supports fast point-to-point, multicast, and broadcast communication of short messages, as an alternative communication model to shared memory. It is used for example for communication to nodes which may not yet have managed global memory. Also, it is a way of containing memory failures which is not possible if global memory is used.

The system knows how to recover global locks taken by failed nodes. Membership services provides support for the notion of a collective system. It relies on global

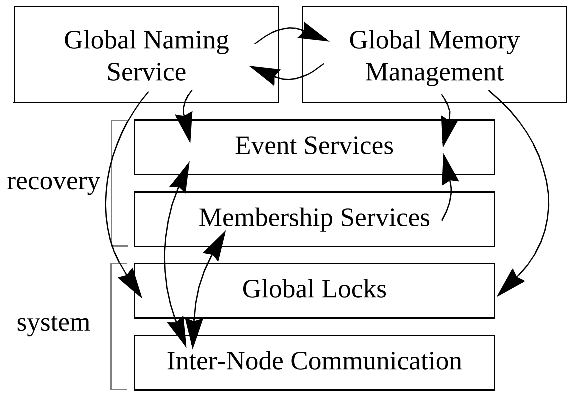

Figure 3 GMM and its Relationship to other Components *(recovery, GNS, global locks, etc.).*

locking and inter-node communication. It supports the software interface to actions which cause loss of membership, typically fault handling. The changes present in the system (both hardware and software) are reflected by global predicate-based Event Services. They notify registered components using callbacks of the type of event that has occurred. Each component is responsible for registering with Event Services its interest in important events. On each occurrence, each component is responsible for reacting to the event, which typically requires recovering the consistency of data structures.

In this way, the MCS system components act as an additional recovery service to allow aware applications and MCS system components to recover. To best ensure recovery, these components are written to expect failures in their operation. In order to further reduce the probability of failure of the recovery service itself, the complexity of these components is minimized.

Using this substrate, GMM implements management of the nodes' combined global shared memory. GMM references regions it allocates by unique identifiers, Global IDs (GIDs). The GIDs are obtained through the Global Name Service (GNS) which also uses the MCS system components. This Global Name Service provides the namespace for sharing objects in an MCS system.

GMM and other MCS components coexist with the host operating system as device drivers which use host OS services and provide a user API through a device driver and an access library in user-space (see Figure 4).

3.2 Physical Memory Management

Management of the global space is provided by GMM running on each node in the system. In order to communicate, these instances share data structures in global memory, protected by global locks. The root of the data structures is the Master Table which maps from GIDs to

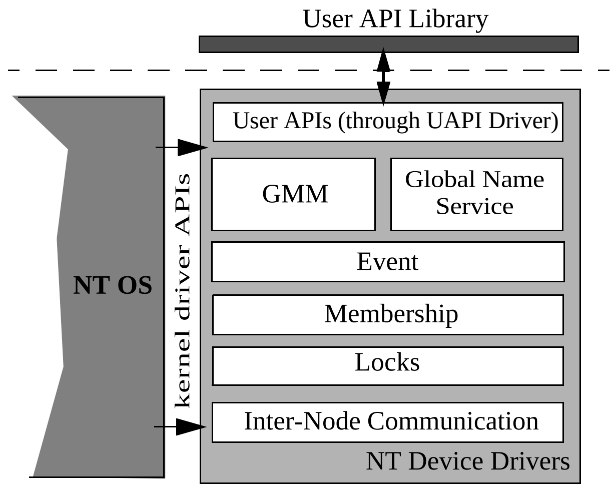

Figure 4 GMM Organization and APIs. *GMM is implemented as a device driver, using the kernel driver interfaces. It exports user and internal APIs for the other MCS components.*

particular allocations of memory, called Sections. Each node has space for a Master Table, but only two copies are used at a time (primary and replica).

Each node then has Section, Sharer and Free memory tables which describe the allocations from global memory in its managed portion of global memory. The Section table describes each region of memory allocated and indicates nodes sharing that Section. The Sharer table then describes which processes (from which nodes) are using memory on this node. Finally, the Free memory list is the usual data structure to hold unallocated memory managed by this node (see Figure 5).

By maintaining two copies of the master table (which are updated on each access) the GMM data structures can always be found upon a single node failure. Other enhancements to these data structures have also been made to ensure the data in the tables can be recovered on a failure (see Section 3.4).

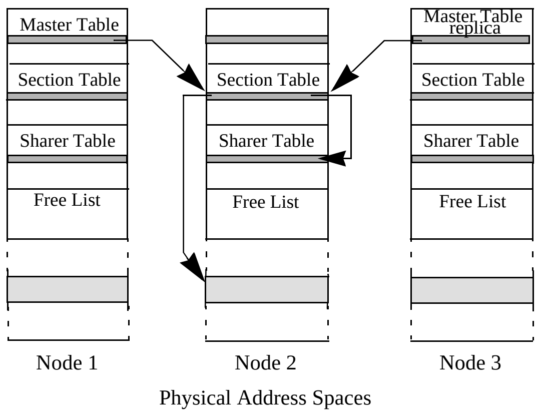

Figure 5 Physical memory management data structures in MCS consist of *the master table (unique for MCS, but replicated), section table, sharer, and free list (per node).*

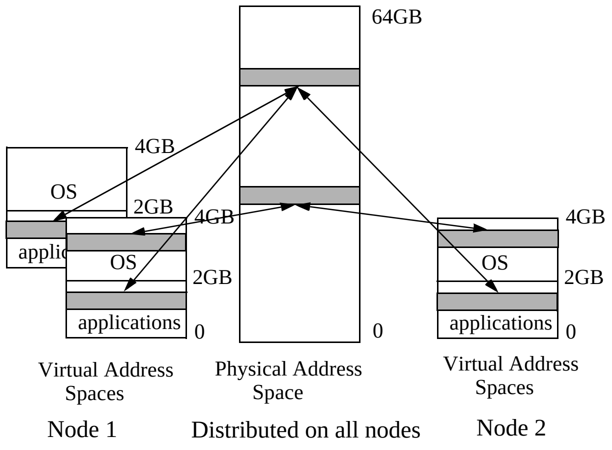

Figure 6 Memory Sharing in MCS: *Applications or MCS kernel components may map the physical global memory into their virtual address spaces and share it with other nodes.*

3.3 Sharing

Applications and MCS system components use GMM to share regions of global memory among themselves. GIDs, obtained from the Global Name Service, provide the naming for these sections. Once identified, these GIDs are passed to GMM either to allocate or share an existing region of global memory. If allocation is requested, then physical memory is reserved using the above structures and mapped into the caller's virtual address space. If a caller opens an existing GID, the desired region is simply mapped into its address space (see Figure 6).

Mapping cannot be performed by using direct control over the virtual memory hardware since the host OS is already using this hardware. For coexistence with the host operating system, it is necessary to be able to use the host OS to map the physical memory into the address space of a particular process. This has the advantage of providing automatic memory protection on a per process basis as long as the host OS supports it.

In the current prototype, global memory is not paged due to limitations of the NT kernel. However, with the ability to add an external page fault handler to the operating system for the global memory, a page system could be implemented. Such a paging system could then take advantage of vacant memory on other nodes to improve performance as a fast backing store before placing copies on stable storage for reliability [11].

3.4 Recovery

The recoverability objective of the GMM design is that surviving nodes be able to recover from the failure of any single operating system on any node in the system. Such failures are detected by a software heartbeat per node monitored by other instances of the membership service in a ring. The failure is signaled using the Event Service to any interested system component. This event causes GMM on each node to recover its data structures to a consistent state by performing the following tasks:

- for remote memory used by the failed node, the failed node is removed from the sharer list of this memory,
- for remote memory that is being allocated/released by the failed node, the operation is completed or aborted,
- if the failed node contained master or replica, a new master/replica is allocated on a surviving node, and
- the free list is updated, if memory is no longer shared.

If the failure also caused loss of the node's resources, e.g. shared memory, then a separate event is issued which causes GMM to:

- remove access to those lost sections and bring its data structures into consistency.
- flush the node's caches.

The biggest recovery challenge comes when an OS crashes during memory allocation. This could result in some of the GMM tables being partially updated or locked but not released. As an example, consider a failure when the node allocating memory fails after identifying a portion of physical memory for use. Since this portion of physical memory is no longer present in the free list, this portion of memory would be unavailable for allocation and hence would be lost.

To overcome this problem, global memory allocation is implemented as a two-phase nested transaction. In the first phase, all the required resources are reserved. After reserving, the necessary data structures are updated. Finally, these resources are acquired. In the case of an inability to acquire a particular resource, the reserved resources are released. During global memory allocation, the following resources are needed:

- A Master Table entry to map the GID associated with this portion of global memory to the identity of the node and the section of the physical memory. This is maintained on the primary and replica for reliability.
- A new descriptor in the Section Table on the node hosting the physical memory to map the section with the physical address. The section descriptor maintains an entry into the sharer descriptor table.
- A new descriptor in the Sharer Table to include the identity of the caller node in the sharer list.
- Physical memory from the Free List to associate as this portion of global memory.

By implementing a two-phase transaction system, new entries to each table are marked as temporary until all the resources can be acquired and all the updates can be made. Only then are the new entries in each table committed. During recovery, all resources that had been reserved by the failed node and not committed are reclaimed. Recovery completes only after all uncommitted resources reserved by the failed node are reclaimed.

Once system components have recovered, applications too can respond to system failure events signaled by the recovery components and use this to increase the availability of their service. Of course, application failures themselves are still the responsibility of the application to detect and handle itself.

3.5 Locking

To ensure consistency using locking, especially in the context of node failures and recovery, the MCS platform was designed to support hardware spinlock primitives. This design allowed the lock subsystem to be completely recoverable from single node or memory subsystem failures. If a node hosting a spinlock failed, it was designed to redirect (in hardware) the lock to a new node and was set to return an error code of further access. Clients using the locks and receiving this error code, would cooperate to recover the state of the lock.

Consider the case that surviving GMM code was holding a lock at the time of the crash. In this case, when exiting the critical section it would receive an error code and could reset the lock state and recover synchronization. If, however, GMM code execution was lost with the lock held, further acquisition would be prevented from accessing the critical section until the recovery code recovered the data structure and reclaimed the lock by resetting it.

Unfortunately no implementation of this hardware locking support was available on any existing platform. So an emulation of the desired semantics was implemented using NT's spinlocks. This emulation provides for error code reporting when the locks are lost and allows lock resetting by the recovery process.

3.6 I/O

I/O may occur during single node or memory failure. Local I/O operations have the same recovery semantics as a traditional NT system, but problems arise when I/O is made to remote systems through global memory. Existing I/O operations to failed memory are taken care by the platform hardware. While client access to the data requires careful use of locking for mutual exclusion and checking for device error codes to avoid consuming erroneous data.

3.7 Application Recovery

Application recovery is the sole responsibility of the application. Applications choosing not to support any form of recovery can either continue regardless (hoping no ill effects will result), respond to signals by the system, or leave restart/recovery to a third party (such as an application monitor).

Applications are signalled a failure event either before or after the system has fully recovered depending on the type of event. If an OS crash caused data inconsistency or hardware was lost, the system recovers first and then signals the application to make itself consistent. However, if resources are manually removed (e.g. shutting down a node for an upgrade) then, after initially informing the system components, the application is signalled first. It is then allowed some time to recover before the system components fully recover from the lost resources. This allows the application to copy existing data and reposition resources before the system attempts to forcibly revoke lost resource allocations. It also improves performance of recovery by eliminating unnecessary recovery of resources the application will release itself.

4 Prototype Implementation

In order to experiment with our approach, a prototype implementation has been created under Windows NT 4.0 Enterprise Edition. The underlying machine is a 4 node multi computer with an SCI interconnect providing hardware coherent shared memory. Each node is a 4-way Pentium II 200Mhz machine with a Gigabyte of memory contributing 256Mb to local memory for the host OS and 768Mb to the global memory pool. NT is informed using the `/maxmem` command in the NT Loader to manage only the lower 256MB region.

4.1 Integration with NT

GMM and other MCS components coexist with the host operating system, running in kernel-mode implemented using the NT Device Driver Kit (DDK) [30].

Each MCS component is written as a separate device driver to provide a modular system design, each exporting an internal API to the other components. A control device driver synchronizes the initialization of the device drivers and the MCS system software.

At boot time, the MCS software on a single node (the primary, defined as the alive node with the lowest *id*) coordinates the boot, allowing each additional node to enter the system one at a time. As each node joins, this event is communicated to other nodes using the fast inter-node communication. This allows the system to build the membership of the system and communicate any events to members. Once these are initialized, the global memory manager initializes. GMM and other MCS components are designed to allow rejoin of failed nodes, even though the current development platform does not allow for this (limitation of the firmware).

GMM starts by initializing empty versions of its tables and then it contacts the current primary node in order to reference the master tables. At this time, GMM has registered itself for OS failures with Event Services. If this node discovers as it joins the system that there is no replica, it copies the master table and informs all other current nodes that it wishes to be the replica. Only one node enters at a time, so there are no race conditions.

Although various MCS components (GMM, GNS, etc.) are implemented as kernel device drivers, they actually function as shared library components in the kernel. A separate MCS component, called the UAPI driver, is registered with the operating system as a driver. It receives requests from user space in the form of *ioctls* (I/O controls), which it translates into procedure calls to the appropriate MCS components. An MCS DLL (Dynamic Link Library) manages all of the *ioctl* communication with the driver, presenting user space applications with an explicit procedural interface.

The UAPI driver also provides generic MCS bookkeeping services. It keeps track of all user space processes that call MCS APIs, notifying MCS components when a process exits so that accurate reference counts can be kept, data structures can be properly cleaned up, and system resources can be recycled and reused.

The UAPI driver also maintains process indexed mappings for all MCS kernel objects (global shared memory segments, global mutexes, and global events) created or opened by any user process. This relieves the individual MCS components from validating and translating the GIDs. For example, a user process provides a GID when it calls the API provided to map a global memory segment into its address space. In responding to this call, MCS must first find a corresponding data record and verify that the given process has access to the corresponding memory segment. Both of these tasks are done by the UAPI driver before calling GMM.

4.2 GMM APIs

The primary goal of the GMM user space APIs is to provide a convenient interface allowing processes on different nodes to access the same memory resources. Included is a strong foundation for different processes to use and maintain identical virtual address mappings to shared memory, regardless of where the processes run.

Maintaining the same virtual address mappings to shared memory is significant because it allows applications to use direct memory references to or within shared data structures. This in turn allows the virtual address for any memory location to serve as an object identifier as well as a memory access handle. A linked list with the links implemented as direct memory references is a typical example. Using virtual addresses in a dual role (for identifiers as well as access handles) is prevalent in Windows NT programming.

Applications need to be organized as multiple processes to take maximum advantage of the availability and recoverability features. The MCS system software is designed to confine the effect of OS failures to a single node. Recoverable applications are expected to do the same. When a node goes down, there must be application processes already running on other nodes in order to recover.

Being able to maintain virtual address mappings to shared data structures through recovery operations in response to faults is an important part of the support provided. Consider a recoverable application maintaining two copies of global shared data in such a way that at any given point in time, one copy or the other is always in a consistent state. Typically, there will be a primary copy that is directly accessed during normal operations, and a secondary copy that is updated only on transaction boundaries of coarser granularity.

When a failure occurs that compromises the primary copy, application recovery uses the secondary to restart computations from a consistent point. This is most quickly facilitated by promoting the secondary to become the primary. However, it requires a change in the virtual to physical memory map for each of the application processes, or that recovery by the application includes repairing all of its memory references to the primary. The latter would be particularly error prone, even for highly disciplined programmers. Moreover, it would pose a significant obstacle for attempts to modularize recovery code from normal operation code. Thus, the GMM user space APIs allow processes to reserve virtual address ranges, which can be freely mapped and remapped to different sections of physical memory. The GMM APIs can be classified into the following groups:

1. Reserving and unreserving virtual address ranges.
2. Acquiring and relinquishing access to identified physical memory resources.
3. Mapping and unmapping specific virtual address ranges to specific physical memory resources.

Keeping these groups independent of each other as much as possible is the key for allowing multiple processes to maintain the same virtual mappings to shared memory. After reserving a given range of virtual addresses and acquiring access to a given segment of physical memory, a specified portion of the virtual space is mapped to a specified portion of the physical memory. This mapping can be undone without losing reservation and portions of it can be remapped to other segments of physical memory.

4.3 Security

Security in MCS builds on the access control mechanisms provided by NT. Each object secured by NT has an associated security descriptor which contains an access control list (ACL). When a user attempts to access an object, the security descriptor is consulted and the access is verified against the ACL. MCS must maintain a globally valid association between each MCS kernel object and its security descriptor. This allows security information to be retrieved even when a user accesses an object (e.g. a shared memory segment) located on another node. MCS subsystems that provide global objects must be modified to perform security checks using this information.

There are two issues involved in managing globally accessible security descriptors: storage and lookup. The security driver stores the descriptors in a table in global shared memory. This table is identified by a GID, and the combination of this GID and an offset within the table make up a globally valid address. This address is then stored with the GID for the protected MCS object, making it possible to retrieve the security descriptor when the object is accessed.

MCS security is implemented by a security driver and some modifications to the subsystems that provide kernel objects. The security driver implements routines to assign a security descriptor to an object and to retrieve the security descriptor for a particular object. Other subsystems are modified to use these routines when creating new objects and verifying accesses.

4.4 Issues with Extending NT for GMM

During our implementation, we encountered three problems with integrating our system with the NT kernel.

Reservation of the virtual address space. Since GMM uses shared data structures one of the most convenient ways to implement the data structures is using pointers. In order to use this optimization we need to have the same virtual memory address across all nodes of the system. Unfortunately, under the NT kernel, there is no way to guarantee the virtual address allocated to the global memory mappings. Instead it is only possible to create mappings as early as possible to hopefully receive the same address. Rather than relying on this ad-hoc solution, the data structures instead are implemented through table indexing. While this is not a great problem, it does reduce the readability of the code base.

Intercepting page faults. Our second problem limited our implementation and consideration of adding paging to our system. It appears that there is no way in the NT kernel to add an external pager for a region of memory. Since our system exists along side NT rather than inside it, Windows NT does not manage the physical memory of the global pool and so we would need to add our own separate pager in order to manage this address space.

Scaling memory beyond 4GB. Windows 2000 supports Address Windowing Extensions (AWE) interfaces for using more than 4GB of physical address space. AWE allows multiple processes to use more than 4GB of physical address space. In addition, a single process can use more than 4GB in a limited way (only 4GB can be mapped at a time, since virtual address space is still limited to 4GB). The AWE interfaces represent a step in the right direction, however, they fall short of the GMM requirements with the inability:

- specify the physical addresses to be mapped to a certain virtual address space: the AWE returns free (non-contiguous) physical pages,
- reserve physical address space for GMM, i.e. NT should not allocate the physical ranges shared between nodes to other local mappings, and
- separate *unmap* from *free*: AWE supports *reserve* virtual address space, *allocate*, *map*, and *free* physical pages; in the absence of inter-node sharing, there is no need for *unmap*, it is achieved as a part of *free*.

New 64-bit processors (e.g. ia64) will relieve some of the problems encountered with designing and implementing GMM. First, the 4GB limitation would go away. Second, because of the large virtual address space, sharing the space among processes would be easier to implement (space could be reserved ahead for sharing purposes). Next, the recovery model will be improved, especially the memory failures.

4.5 Limitations of the GMM Prototype

The current GMM prototype implementation under NT has the following limitations:

Premapped and non-paged global memory. All global memory is pre-allocated in the NT non-paged system virtual address space and subsequently allocated from this pool, since that is the only way NT will permit dynamic mappings of the memory into the user portion of a process' address space.

Incompatibility with the local OS semantics. Examples include address space inheritance, security, etc. This limitation is introduced because the local OS is not managing GMM memory and it is not in the position to handle it in accordance with the GMM requirements. In order to make this possible, the GMM interfaces need to be used to achieve security, sharing, recovery, etc.

Inability to test memory failures. Given the reliance of the current hardware platform on an SCI ring, it is difficult to test for memory failures as a result of an entire node failure rather than a simple OS crash. A node failing cannot be simulated by breaking the SCI ring to isolate a node without disrupting communication to all nodes. Instead, we simulated memory failures by explicitly unmapping memory which would in normal operation be mapped as part of global memory.

5 Using GMM

5.1 Shared Memory Programming Models

An MCS global application is a set of one or more cooperating processes that run on the nodes of an MCS system. There may be multiple processes per node, and the processes may be multi-threaded. MCS global applications that have been modified to take advantage of an MCS system use the global memory to achieve two benefits: performance scalability and high availability. By performance scalability we mean that the throughput of an application should increase in proportion to the amount of computing resources allocated to it. For example, an MCS application which has its processes running on two nodes of an MCS system should deliver roughly double the throughput of an application running on one single node. In this context, high availability means that the application can continue to provide service to users in the event of OS (and in future hardware) failures on any node in the system.

An application must have certain characteristics to be able to exhibit performance scalability while using GMM. The first characteristic is the same as in the case of an SMP system: the application must consist of independent threads, and the throughput of the application must increase with the number of concurrent threads. The shared data set can be placed in global shared memory where it can be accessed by all processes comprising the global application. There are two keys to achieving high availability. First, each process must maintain its state in global shared memory, so that if a process terminates (for example, due to the crash of the operating system on its node) then the task it was executing can be completed by another process on another node. Second, each data item stored in global memory should be backed up by the application to a redundant copy, either in global memory residing on a different node or on disk. If memory is lost at a result of a node failure then the redundant copy can be referenced.

5.2 Kernel Components that Use GMM

MCS kernel components can use global shared memory to their advantage. In Windows NT (and other OSes), the kernel and privileged-mode device drivers share a common virtual address space. While the Windows NT kernel itself does not use global shared memory, it permits privileged-mode drivers to map global memory into the kernel address space. Although all kernel components can access all mapped global sections, the common practice is for MCS kernel components to share sections only with their counterparts on other nodes.

GMM uses its own services: the tables describing the global memory sections attributes and the locations of unallocated ranges of the global memory are themselves stored in global shared memory. This permits distributed management of the global memory resource. GMM on any node can make a new global memory allocation by updating the shared tables.

Global networking relies on global shared memory as its physical transport medium. This component presents itself to the operating system as a standard networking driver, so all standard networking protocols are supported. To send a packet from one MCS node to another, the sender places the packet in a buffer in a global memory section and posts an interrupt to the receiving node. The global memory sections required to hold the buffers are allocated when nodes initialize their networking. Each node creates a global memory section of about 2 MB on each other node to which it can send packets. For each buffer, there is only one node that writes and only one node that reads the contents.

Another MCS kernel component implements global synchronization objects, mutexes and events, for use by user space applications. These objects have the same semantics as the Win32 mutexes and events, except that they can be used from any process on any node. These objects also have recoverability features allowing them to survive node and resource failures in the system.

The final example of a kernel component using global shared memory is the global file system. This file system presents a common global file tree across all MCS nodes. The file system maintains a cache of recently used file blocks in global shared memory, so that performance is increased for accesses to shared files (opened by multiple processes simultaneously). The implementation of the global file cache has not been completed, so performance statistics are not available.

5.3 Applications that Use GMM

An application whose characteristics should benefit from the properties of global memory management is a database manager. Most database managers can increase their throughput in proportion to the number of concurrent threads accessing (different parts of) the data set. In current practice, it is common to have data sets many gigabytes in size, and many workloads feature a significant number of update operations. We considered porting a major database manager, such as Oracle or Informix, to MCS, but the size and complexity of the code base would make this a difficult undertaking.

The first global application that is ported to our prototype MCS system is a main-memory database manager from TimesTen Performance Software. This product supports the same query and transactional update functionality of a conventional database manager, but with increased performance and predictability of response time owing to the fact that the entire data store resides in main memory. The structure of the application makes it simple to place the data store in global shared memory and to synchronize access to it from multiple nodes.

The second MCS global application is a Web server modified to share Web caches in global memory. This application was chosen to allow future study of availability and performance issues in potential future Internet systems. Both the Microsoft IIS and Apache Web servers were modified to manage a cache of recently-used files, rather than relying on the file system's cache. The cache is placed in global shared memory so that each node could read cached files and copy new files from the file system into the cache. A Least-Recently Used (LRU) global cache management algorithm was used. Results in this paper represent initial demonstration figures of this work using the Microsoft IIS Web server only.

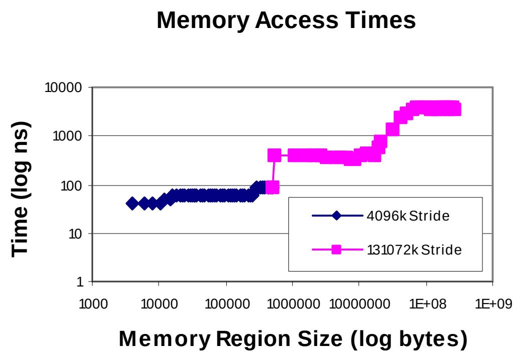

Figure 7 Memory Access Times on the Prototype: *Two memory region tests cover the entire memory hierarchy.*

6 Experiments

In order to obtain the baseline performance of the prototype configuration, we used lmbench [20] to measure the latency of the various levels of the memory hierarchy. Lmbench calculates the minimal memory access time by traversing the memory region at various step distances (strides) to determine cache effects. By using two memory region size ranges and two strides to best determine cache effects (4Kb-512Kb with access stride 4096 bytes and 512Kb-256Mb with access stride 128k) we were able to effectively evaluate the complete memory hierarchy access latency, including first and second level caches as well as local and remote memory. This is achieved by using memory list structure those elements are placed at a certain stride through a memory section (larger than the cache size) from local and shared memory. In our experiment, we measured the latency for the level 1 cache latency to be 40*ns*, the level 2 cache latency to be 50~85*ns*, non-cached local memory (both local DRAM and the local SCI cache) to be about 304~314*ns*, and the remote memory load access latency to be approximately 3950-4125*ns* (see Figure 7).

Another set of performance measurements compares the use of networking over shared memory v. loopback. The results are presented in the table below. Shared memory networking demonstrates relatively good performance since large blocks of data are being transferred.

Measurement v. Environment	**Loopback** *(Kb/sec)*	**Shared Memory Network** *(Kb/sec)*
FTP Transfer: 30MB (binary)	*9286*	*7870*
TTCP Transfer: 2K writes of 8KB	*10233*	*8197*

We measured the performance of the MCS version of the TimesTen database manager by using a debit/credit benchmark patterned after TPC-B. Our data set had

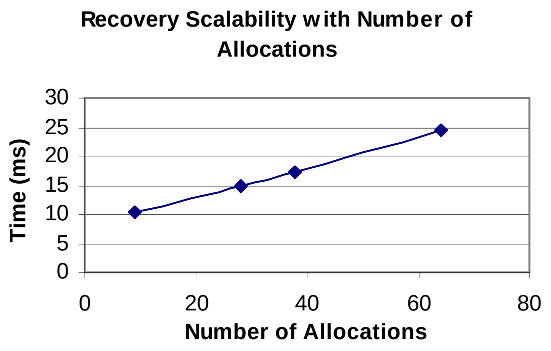

Figure 8 GMM Recovery Scalability: *Recovery was timed on the four nodes when one failed with an increasing quantity of shared memory allocations.*

90,000 rows. Our first experiment compared the released version of TimesTen with the MCS version on a uniprocessor with the data set stored in memory local to the node. This showed that the overhead of replacing native Windows NT functions for memory management and process synchronization with the corresponding MCS functions was approximately 1%. Our second experiment compared the MCS version on a uniprocessor with the data set stored in memory local to the node with a similar configuration with the data set stored in remote memory. The two runs showed virtually identical performance, showing that our data set fit in the system's remote memory cache of 32 MB, whose response time was the same as for local memory. Finally, we compared the released version on a four-way SMP with four MCS nodes, and again the performance was similar, because the execution time was dominated by contention for the lock on the data set.

Experiments on the recoverability of the GMM modules on three nodes in presence of the failure of the fourth node were performed. These experiments measured recovery time of GMM modules in the MCS system and scalability with number of allocations of GMM recovery. In our experiments we fail one node (blue screen it) and the other three nodes successfully recover. Our experiments showed that the time to recover GMM takes approximately 10.2*ms* for one shared application memory region (plus 8 taken by the MCS system components). This represents a significant improvement in service disruption time compared to the time to reboot the sharing nodes. Experiments also showed that as these were scaled to a reasonable maximum of 64 shared allocations, the overhead to recover scaled linearly (see Figure 8). These figures include the time to signal the failure, recover on all remaining nodes and resynchronize on completion.

Initial experiments with a Web Server application were performed to demonstrate the use of the GMM modules in a real world application. Using two nodes of our prototype, we ran the Microsoft IIS Web server with a modified shared memory cache. Then a popular web benchmark was run to provide a workload benchmark for these two web server machines. Four client machines were used to generate sufficient workload to maximize the work of the servers, with the total workload spread across the two server machines. Each client ran four load-generating processes. Results were measured for just one machine with a 200Mb Web cache and two machines each contributing 100Mb to a shared GMM cache. The second configuration therefore has the same total cache size as the first configuration, but with the use of shared memory between machines has allowed the application to be scaled to two machines while maintaining any sharing. Results are presented in Figure 9.

Our experiments measured that the performance approximately doubled when processing and I/O resources were doubled. Such speedups are common with web servers since content can be replicated, but this requires doubling the memory resources also. In this experiment, the servers seem to scale with server resources while memory resources remained constant due to the sharing of resources. These figures, while interesting, are not designed to demonstrate specific performance benefits, since they are relative and entirely unoptimized. They do, however, demonstrate that the MCS system and in particular the GMM modules are capable of running real world applications and provide suitable potential for the real world use of such a platform. Once this work has been completed, we hope to publish further results.

7 Lessons Learned

1. It is possible to extend NT with global memory management without changes to the existing code base. However, this is only true for limited implementations, where memory is preallocated and pinned.

2. In order to achieve fully functional GMM there is a need for extensions to Windows NT (see Section 4.4

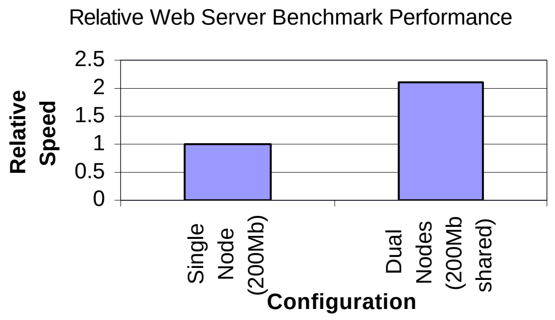

Figure 9 Web Server Benchmark Performance: *Throughput in single- and two-node shared memory configurations.*

for more details). If NUMA-like machines become widely spread, this effort is likely to be standardized among OS and hardware vendors.

3. It is hard to develop a flexible and easily deployable programming model acceptable for legacy applications. Applications need to be parallelized in order to allow for easy deployment using GMM. Kernel components are more likely to use GMM since their use is hidden from the users. An example is networking (see Section 5.2).

4. Recoverable programming models for GMM are even harder. They require careful design and replication of data structures. Furthermore, they are error-prone since recovery from partial updates can be complex. Memory failures (not addressed in the initial implementation, but considered long term) make this problem even harder and more hardware dependent.

5. Recovery poses a requirement to the local OS “not to get in the way”. For recovery purposes, it is easier to provide extra code that pays attention to preserving the state, rather than relying on the existing (nonrecoverable) OS which may lose the state on the OS execution stack. This may duplicate functionality, but the same code can be used for different OSes.

6. Hardware support for containment and recovery is very important. We realized that in order to support recovery from hardware failures we need additional hardware support in order to notify and mask memory failures, which would normally cause the system to machine check and fail.

8 Related Work

There are a number of systems related to GMM both in academia and in industry. In academia, most related to our work are OSes developed at Stanford: Hive [6], DISCO [4], and Cellular DISCO [13]. For Hive, Teodosiu explored the possibilities hardware fault containment in multiprocessor systems [28]. Cellular DISCO is derived from the DISCO virtual machine work. DISCO showed that the scalability limitations of multi-processor systems could be overcome by running multiple virtualized OSes rather than scaling one OS. Cellular DISCO takes this virtualization further by using the ability to kill and restart virtualized OSes for fault isolation and containment. This approach does not consider recovery other than rebooting virtualized OSes. Our approach attempts to provide the ability to recover from software and hardware failures.

There has been a lot of work on memory management for early NUMA systems [3, 9], as well as for NORMA (distributed memory) model [1, 2, 12, 31], but none addressed failures. The cooperative caching project at the University of Washington investigated the use of memory from underutilized workstations [11]. Some of distributed shared memory systems address fault recovery [5, 17]. The Rio system addresses recoverability of the OS from SW failures, including wild writes [7].

Many companies provide 'high availability' systems through partitioning, such as Sun's UE10000 [27], Unisys's Cellular Multiprocessing Architecture [29], Sequent's NUMA-Q servers [26], Compaq's Wildfire/OpenVMS Galaxy platform [8, 10], SGI’s Cellular IRIX /SGI 2000 family [19] and IBM's S/390 Parallel Sysplexes [21, 24]. These systems provide increased availability by hardware partitioning, redundancy, and by running in “lock-step” (IBM's Sysplexes). These systems rely on hardware features to allow failures to be contained per partition (a logical node or set of nodes). By having such partitions and executing multiple OS instances, they provide the ability to contain the effects of software failures while since allowing shared-memory between instances for fast communication. On such systems, high availability software provides error reporting/logging and control of partitioning where applicable. But non-redundant software and hardware failures cause failure of particular partitions and are resolved by rebooting. Our work tries to increase the availability envelope by using software which can attempt to recover from dependencies on software crashes rather than requiring dependent partitions to reboot. This form of recovery is of increased importance when resources, such as memory, are being shared and partitions are therefore more tightly coupled. This type of sharing, as typified by common applications, such as Oracle's Database Server, is key to obtaining good performance [23].

9 Summary and Future Work

We designed and implemented a prototype implementation of global memory management for the NT OS. We achieved this without modifications to the OS. However, the prototype implementation has limitations, such as non-paged global memory. In order to remove these limitations, some modifications to the underlying OS are required. We described the required functionality missing in the existing NT implementation for fully functional GMM. We described GMM recovery as well as some applications that we used with it. Finally, we derived lessons learned.

Our future work will address memory hardware failures and specifically how to recover from them in case of the ia64 architecture. We believe that most of the failures that will remain will be due to software [14]. However, with the increased high availability requirements of scalable systems, the mean-time between failure of memory and other components, such as interconnections, processors, will increase. This may not be sufficient for systems such as enterprise data servers. Therefore we need recoverable programming models to fill the gap. In particular, we are interested in the tradeoffs between hardware and software support for optimal recoverable programming models.

Acknowledgments

Tung Nguyen is the father of the MCS program. Lu Xu implemented GNS and parts of GMM. Todd Poynor and Guangrui Fu conducted Web server and GMM latency experiments. Chen Li ported TimesTen to MCS. Jork Löser evaluated GMM on Linux. We thank TimesTen for letting us use their product. We are indebted to Herman Härtig, Keith Moore, Todd Poynor, and Mike Traynor for reviewing the paper. Their comments improved contents and presentation.

References

[1] Abrosimov, V., Armand, F., Ortega, M., I., “A Distributed Consistency Server for the CHORUS System”, *Proc. of the USENIX SEDMS*, March 1992, Newport Beach, pp 129-148.

[2] Black, D., Milojicic, D., Dean, R., Dominijanni, M., Sears, S., Langerman, A., “Distributed Memory Management”. *Software Practice and Experience,* 28(9):1011-1031, July 1998.

[3] Bolosky, W., Fitzgerald, R. P., Scott, M. L., “Simple but Effective Techniques for NUMA Memory Management,” *Proc. 12th SOSP*, pp. 19–31, Wigwam Litchfield Park, Az, December 1989.

[4] Bugnion, E., Devine, S., Rosenblum, M, “Disco: Running Commodity Operating Systems on Scalable Multiprocessors,” *Proc. of the 16th SOSP*, Saint Malo, France, pp. 143-156, Oct. 1997.

[5] Cabillic, G., Muller, G., Puaut, I., “The Performance of Consistent Checkpointing in Distributed Shared Memory Systems”, *Proc. of the 14th Symposium on Reliable Distributed Systems*, Bad Neunahr, Germany, September 1995.

[6] Chapin, J., et al., “Hive: Fault Containment for Shared-Memory Multiprocessors,” *Proc. of the 15th SOSP*, pp. 12-25, Dec. 1995.

[7] Chen, P.M., et al., "The Rio File Cache: Surviving Operating System Crashes", *Proc. of the 7th ASPLOS, October 1996.*

[8] Compaq, “OpenVMS Alpha Galaxy Guide”, Downloaded January 2000, http://www.openvms.digital.com:8000/72final/6512/6512pro.pdf.

[9] Cox, A., Fowler, R., “The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum,” *Proc. Twelfth SOSP*, pp. 32–44, December 1989.

[10] Digital Readies 500mhz-based 'Wildfire' Server”, PC Week, March 1997.

[11] Feeley, M.J., et al., “Implementing Global Memory Management in a Workstation Cluster”, *Proc. of the 15th SOSP*, Dec. 1995.

[12] Forin, A., Barrera, J., Sanzi, R., “The Shared Memory Server”, *Proc. of the Winter USENIX Conf.,* San Diego, 1989, pp 229-243.

[13] Govil, K., et al., “Cellular DISCO: Resource management Using Virtual Clusters on Shared-Memory Multiprocessors,” *Proc. of the 17th SOSP*, pp. 154-169, December 1999.

[14] Gray, J., and Reuter, A., “Transaction processing: Concepts and Techniques,” Morgan Kaufmann, 1993.

[15] Intel, "Intel Architecture Software Developer's Manual", and "Addendum”, 1997.

[16] Intel, "The Intel Extended Server Memory Architecture”, 1998.

[17] Kermarrec, A-M., Cabillic, G., Gefflaut, A., Morin, C., Puaut, I., “A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability”, *Proc. of the 25th Int’l Symposium on Fault-Tolerant Computing Systems,* pp 289-298, June. 1995.

[18] Lampson, B., “Hints for Computer System Design”, *Proc. of the 9th SOSP*, October 1983, pp 33-48.

[19] Laudon, J., Lenoski, D., “The SGI Origin: A ccNUMA Highly Scalable Server”, *Proceedings of the 24th International Symposium on Computer Architecture,* pp 241-251 June 1997.

[20] McVoy, L., Staelin, C., “lmbench: Portable Tools for Performance Analysis”, Proc. of USENIX 1996 Conference, San Diego, CA, January 22-26, 1996, pp 279-294

[21] Nick, J.M., et al., “S/390 Cluster Technology: Parallel Sysplex”, *IBM Systems Journal,* v 36, n 2., 1997.

[22] Oracle, “Oracle Parallel Server in the Digital Environment, High Availability and Scalable Performance for Loosely Coupled Systems,” Oracle White Paper, June 1994.

[23] Oracle 8 Support for the Intel^R “Extended Server Memory Architecture: Achieving Breakthrough Performance,” *Intel Document, Oracle's Note,* 1998.

[24] Pfister, G., “In Search of Clusters”, Prentice Hall, 1998.

[25] Popek, G., Walker, B., “The Locus Distributed System Architecture”, *MIT Press Cambridge Massachusetts,* 1985.

[26] Sequent White Paper, “Application Region Manager”, Downloaded in January 2000 from http://www.sequent.com/dcsolutions/agile.pdf.

[27] Sun Microsystems, “Sun Enterprise^{TM} 10000 Server: Dynamic System Domains”, White Paper, Downloaded January 2000, http://www.sun.com/datacenter/docs/domainswp.pdf.

[28] Teodosiu, D., et al., “Hardware Fault Containment in Scalable Shared-Memory Multiprocessors,” *Proc. of the 24th ISCA*, June 1997.

[29] Unisys, “Cellular MultiProcessing Architecture,” White Paper, Downloaded January 2000, http://www.unisys.com/marketplace/ent/downloads/cmparch.pdf.

[30] Viscarola P. and Mason, W.A., “Windows NT Device Driver Development,” *Macmillan technical Publishing,* 1999.

[31] Zajcew, R., *et al*., “An OSF/1 UNIX for Massively Parallel Multicomputers”, *Proc. of the Winter USENIX Conference,* January 1993, pp 449-468.