Application Programming on a Shared Memory Multicomputer

Todd Poynor, Tom Wylegala
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-114
September 6th, 2000*

E-mail: poynor@hpl.hp.com,
wylegala@hpl.hp.com

shared memory, fault containment, recovery, memory failures, multicomputers

We present our experience investigating issues involved in writing applications for a multicomputer comprised of computing nodes coupled through global memory. Of primary interest is *fault containment*, preventing faults in one component from spreading to other components, and the means by which applications can increase their availability using shared state to aid in recovery from failures. Our architecture pursues application-level fault containment as strong as that of shared-nothing clusters within a shared-almost-everything environment. Applications freely employ a variety of global resources and explicitly recover from failures. We examine the challenges of programming in such an environment and investigate support our platform could provide to aid developers. We demonstrate these issues using a Web server file cache and present some performance scalability analysis. We also examine business issues related to the acceptance of such a platform in the commercial application marketplace.

* Internal Accession Date Only Approved for External Publication

Application Programming on a Shared Memory Multicomputer

Todd Poynor
HP Laboratories
poynor@hpl.hp.com

Tom Wylegala
HP Laboratories
wylegala@hpl.hp.com

1. Introduction

This paper describes our experience to date in the HP Labs MultiComputer Systems (MCS) project investigating issues involved in writing applications for a shared memory multicomputer, with an emphasis on fault containment for various types of hardware and software failures. Shared memory multicomputers hold considerable promise for the commercial marketplace as modular architectures that transcend SMP scaling bottlenecks while preserving SMP-like memory load/store programming models. Many multicomputer platforms today do not fully deliver these benefits because most resources are partitioned and shared memory is limited to inter-node communication via message passing, as in a shared-nothing cluster. Those multicomputer platforms that we are aware of that do allow access to global resources have limited support for containing faults from propagating across nodes, leaving multicomputers at a disadvantage when compared to conventional clusters in this regard.

The MCS project is, in part, an experiment in pursuing fault containment as strong as that of shared-nothing clusters within a shared-almost-everything multicomputer. We examine the challenges of programming in such an environment and investigate support our platform could provide to aid developers. We demonstrate these issues using a Web server file cache application and present some performance scalability analysis of the application. Additionally, we also examine business issues related to the acceptance of such a platform in the commercial application marketplace, describing consultations with potential vendors of multicomputer software.

The remainder of the paper is organized as follows: section 2 introduces the MCS multicomputer platform, multicomputer applications, and the fault containment problem; section 3 describes our recoverable programming model; section 4 presents our Web cache example application coded to that model; section 5 summarizes consultations with application vendors who considered writing applications for our platform; section 6 presents related work; section 7 describes some potential future directions for our work; and the final section presents conclusions.

2. Shared Memory Multicomputers and the MCS Platform

This section introduces shared memory multicomputers, the MCS platform, and multicomputer applications.

2.1 Shared Memory Multicomputers

A shared memory multicomputer is defined as a set of **N** computers, each autonomous in the sense that it can run its own independent operating system, tightly coupled together through a global shared memory resource. The fact that each computer, or **node**, can operate independently implies that each has its own set of resources (such as processors, memory, and I/O). These resources are called **local** or **private** to emphasize the fact that they are accessible only by the one node that owns them. The Global Shared Memory (GSM) resource is symmetrically accessible from all nodes, through a channel whose bandwidth and latency characteristics are roughly comparable to those for local memory, such that most applications would not need to be written to account for memory page placement. This architecture is distinguished from typical SMP and single-kernel ccNUMA systems by the fact that it supports multiple independent operating system instances, and from a shared-nothing cluster by the fact that communication among nodes occurs via loads and stores rather than message passing.

Note that the partitioning of a multicomputer into nodes and the accessibility of local and global resources does not imply any particular physical organization of the system. Figure 1 shows one possible organization for a multicomputer with three nodes:

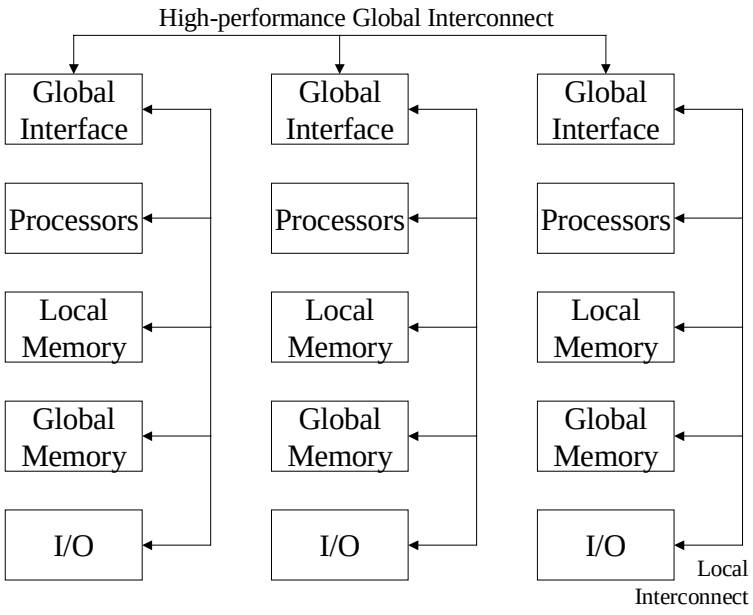

Figure 1: Multicomputer with resources contributed by each node

In such a multicomputer, each node has a number of resources that may be accessed locally without using the global interconnect. Local resources include a contribution to the GSM pool, and thus each node has non-uniform access to the various GSM areas. Failure of a component

likely results in a particular node being damaged or isolated from the others, or in all nodes being isolated from each other. Figure 2 depicts another possible organization:

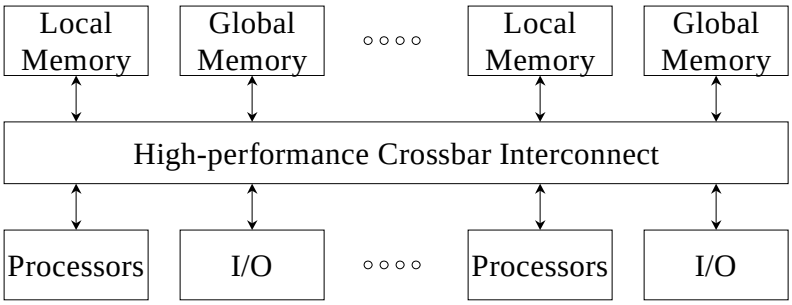

Figure 2: Multicomputer with all resources accessed via interconnect

In this "dance hall" [20] organization, failure of the interconnect implies complete failure of the entire system, as also occurs in SMP architectures.

2.2 The MCS Multicomputer Architecture and Prototype

MCS is an architecture specification and prototype implementation of a shared memory multicomputer developed at Hewlett-Packard Laboratories.

A multicomputer is comprised of a number of nodes. Each node is an independent SMP computer containing the maximum number of processors per node that exhibits good scalability. It would be typical for each node to contain at least 1 GB of private memory for the host operating system and for applications that have not been made into global applications. The platform would typically contain at least 1 GB of shared memory per node, which is used by global applications to store global state using an SMP programming model and also serves as a high-speed inter-node communications channel.

The MCS prototype platform is based on a commercially available ccNUMA machine with custom firmware. The prototype features 4 nodes arranged in a configuration similar to that of Figure 1. Each of the four nodes is a four-way SMP comprised of 200 MHz Pentium Pro processors. The number of processors is well matched with the Windows NT 4.0-based MCS prototype operating system, which exhibits good scalability through 4 processors. The global interconnect is a redundant pair of IEEE Scalable Coherent Interface (SCI) daisy chains. Each node possesses 256MB local memory and 768MB GSM.

The MCS architecture is operating system neutral. The prototype implementation uses an unmodified Windows NT 4.0 as the operating system on all nodes, extended through dynamically loaded kernel modules to support multicomputer interfaces. It is a key MCS design point to leverage commodity operating systems with few or no changes, and thus clear distinctions must be drawn between the local operation of the base OS and the global operation of the MCS extensions. For example, process scheduling decisions, management of non-GSM

memory, and so forth use the local policies provided by the base OS, whereas MCS extensions are involved in the management of GSM and other global resources. The resulting architecture leverages commodity operating systems executing in an SMP environment on a single node appropriately scaled for that OS, on top of which applications may transcend the node boundary using global interfaces to achieve the benefits described in the following section.

MCS global memory management and interactions with other MCS kernel components are described in [2]. The architecture of the planned system interconnect is described in [14]. Other MCS shared resources include global I/O using I2O intelligent interfaces, a global file system symmetrically accessible from all nodes, hardware locks that implement atomic read-modify-write operations, and a variety of control registers, such as mailboxes to pass small messages between nodes.

2.3 MCS Global Applications

A global application is a set of processes distributed across the nodes of a multicomputer that cooperate to provide a service. There may be multiple processes of such an application per node, and each process may be multithreaded. The various processes of the application may be fully symmetrical peers, or they may be arranged in master/slave fashion. Each process has access to the local resources on its node through native platform APIs, and to global resources through global APIs.

2.3.1 MCS Global APIs

The global APIs provide natural extensions to local programming abstractions, so that cooperation among the processes and threads of the global application is similar to the cooperation required among threads on an SMP. Communication via message passing is also possible. It would be a natural choice to specify the global APIs to follow conventions similar to the APIs of the host operating system. The MCS prototype is developed on Windows NT, so the MCS APIs are patterned closely after analogous Win32 APIs. MCS APIs were also developed for a Linux platform, patterned after native Linux APIs, showing that the programming paradigms are independent of the exact API specification.

MCS global APIs currently provide access to such resources as:

- GSM objects, called global memory *sections.* Each section is a subset of the system's global memory resource. An application is given interfaces to create a global memory section (allocating memory from the global memory pool), to open an existing global memory section, to map a subset of a global memory section into its virtual address space, and so forth.
- Global synchronization objects, such as "mutex" mutual exclusion objects and globally signaled event objects. These objects allow threads running on different nodes to synchronize with each other. An application is given interfaces to lock a global mutex (blocking until it is available), to release a global mutex lock, to wait for a global event signal, and so forth.

2.3.2 Motivation for Multicomputer Global Applications

The primary advantage of multicomputing from the standpoint of application programming is that the structure of the platform permits the development of applications that exhibit high degrees of performance scalability and availability.

Performance scalability means that an application is able to deliver throughput in rough proportion to the amount of resources (processing, memory, and I/O resources) provided by the platform. The context is that there is an extremely large pool of clients requesting service from the application, and its throughput is the number of service requests processed per unit time. The platform with the best performance scalability is the one that gives the best proportional increase in throughput when more resources are added. The performance scalability of a multicomputer, which has **N** independent instances of the operating system, should be greater than that of an SMP, shared nothing cluster, or traditional ccNUMA system containing the same resources. One reason for this is that contention for locks protecting kernel data structures tends to limit OS scalability in an SMP or single-kernel ccNUMA (many references, for example, [5, 21, 22]). In a multicomputer, the **N** operating systems operate independently in parallel, allowing a multicomputer application to avoid OS scaling bottlenecks and possibly scale to more processors than could be scaled to by a single SMP OS. Other SMP resources that may be application scalability bottlenecks, and that multicomputers and ccNUMA platforms effectively scale up, include the number of processors to which the platform scales and I/O and memory busses [6, 20]. In a shared nothing cluster, application scalability may be limited by the partitioning of resources and by communication among nodes, which sends messages through an I/O channel considerably slower than memory. In a multicomputer, communication is effected through loads and stores that operate with the bandwidth and latency characteristics of memory.

Availability means that the application is able to deliver service even in the presence of failures in platform software or hardware. A multicomputer is able to deliver high availability because an application can be structured to run on multiple nodes simultaneously, such that the parts of the application running on surviving nodes can continue to deliver service even if one node fails catastrophically. A failure in a cluster, such as the failure of a kernel thread, can usually be isolated to a single node, while such a failure is nearly always fatal to an SMP and a single-kernel ccNUMA. A multicomputer shares the robustness of the cluster, with the additional advantage that the faster communication permits employment of a wider range of recovery strategies, but at a price of using recoverable programming models and execution on platforms that avoid single points of failure. It is this fault containment aspect of multicomputer applications that we emphasize in this report.

3. MCS Application Fault Containment

In this section, we present the application support for fault containment and recovery in MCS. We begin with the fault containment goals we wish to achieve for future platforms, followed by the classes of failures targeted for our prototyping work, and then describe the MCS features provided in support of recoverable applications.

3.1 MCS Application Fault Containment Goals

The mean time to failure of a multicomputer decreases as the number of nodes increases. A challenge for multicomputer platforms is to prevent faults in one component of the system from bringing down other parts of the system or the entire system. A shared-something cluster exposes the risk that faults will propagate over the interconnect to introduce faults in other components. For applications that communicate over shared memory, various types of hardware failures that prevent access to GSM and software failures that corrupt the values of shared data structures can cause numerous processes of the global application to fail, severely limiting availability of the application. This issue remains a thorny problem in the industry, particularly when commodity hardware and operating systems are leveraged.

MCS strives for a very high degree of inter-node fault containment and application availability within a shared-almost-everything model for efficiency of resource allocation and scalability. MCS targets truly global applications employing a variety of types of global resources in large numbers and with numerous interdependencies throughout the complex, without increasing the chances of failure of a process due to failures of remote resources. The environment is the highly dynamic commercial data center where the goal is to provide continuous and scalable application service. Processes join and leave the global application at any time due to redeployment of resources, as in response to a change in load or due to failure and reintegration of components. Applications access shared resources in a "safe" fashion, detecting and recovering from failures and reconfiguring when nodes and applications are redeployed. All of this comes at a price of extending commodity operating systems and customizing applications for fault containment, as well as extending commodity hardware platforms to support containment and avoid single points of failure.

3.2 Targeted Failure Classes

Our prototyping work concentrated on recovery from process and kernel termination failures, where one or more global application processes suddenly cease execution due to reasons such as: a process being aborted by the OS in response to a fault or operator request; an OS crash or untimely reboot; and hardware failure or power failure of the associated node. OS crashes were targeted as relatively high probability failures that exercise both kernel and application layer recovery.

Global applications are intended to recover from memory failures on future MCS platforms. "Memory failures" in this context can refer both to memory access that times out due to malfunctioning or powered off hardware and to memory access that incurs a transient, parity-type error that cannot be automatically corrected by the memory hardware. We distinguish between "failed memory" that is no longer available due to hardware failure or power off and "inconsistent memory" that continues to be available but that may be in an inconsistent state. Inconsistency may be introduced by other failures in the system. For example, one node of a multicomputer system fails while holding a remote GSM cache line private, such that updates by that node to the remote node's memory may be lost or only partially flushed from its caches. Our MCS prototype system cannot survive memory access failures, but we have partially prototyped the application-level support for such a system in the future.

3.3 Global APIs for Fault Detection

The section describes the global APIs and system support for fault detection.

3.3.1 Detection of Process and OS Termination

In our work so far, application-level detection of and recovery from both events of sudden termination of a global application process and crash of a remote OS are handled identically. Accordingly, we have relied on process termination detection to also handle the case of OS crashes, without the need to distinguish between the two at the application level.

Process terminations are detected in the application by an indication from the global synchronization APIs that a global mutual exclusion object (mutex) was "abandoned" by the process that previously locked the object without properly releasing it. This suggests that a process crashed, perhaps during an update to global state. The global mutex abandonment is indicated to the application in analogous fashion as that of the native NT interface: a MUTEX_ABANDONED status is returned to an application that calls an MCS routine to lock the mutex.

This interface may also be used to proactively detect process and OS crashes as soon as possible by associating one locked mutex per process in the global application. Each process has a thread waiting to "lock" the mutex for each remote process. These threads wake up upon abandonment of a mutex, at which time the threads signal that application recovery is required.

The underlying MCS operating system components detect mutex abandonment due to process termination using the usual NT local mechanisms, communicated globally to a remote waiter. Mutex abandonment due to OS crash is detected using MCS cluster membership services that watch for missed "heartbeats" communicated over shared memory.

3.3.2 Detection of Memory Failures

We have prototyped some application-level support for future recovery from GSM access failures, in order to better understand how such support affects application programming.

Preliminary MCS designs have discussed interconnect hardware that could play a part in allowing systems to survive memory access failures from commodity processors. These designs specify the following two properties of GSM, among others:

1. Memory for which failed access is recoverable is explicitly designated as such, and applications take special measures to access that memory safely. In MCS, only GSM behaves this way, whereas failures in local non-shared memory continue to be catastrophic to the process or to the system (in keeping with the goal of fault containment).
2. Failed read operations to GSM are satisfied with a preset "dummy" value by special hardware, if required by the processor on which the system is based. This is performed in order to avoid possibly unrecoverable exception conditions that are incurred on present-day processors when a memory read is not satisfied within the required period of time, such as an IA32 Machine Check Abort (MCA).

For our prototyping, we have presumed the existence of a mechanism by which applications may be informed that a previous memory access may have failed. This mechanism could be either that of an exception or signal delivered to the process or an inline call to a function that returns an indication of possible failure. Under either method, applications must check whether previous memory accesses may have failed at appropriate times. These checks must generally occur after access to GSM and before continued execution without recovery from the failure would cause erroneous behavior. For example, if the value read from GSM is a pointer in the C language sense (that is, a memory address) then the application must ensure that the access successfully returned valid contents of the GSM housing the pointer before attempting to dereference the returned value. Much of the application code written for MCS is structured in this fashion in order to study the impacts on development of multicomputer applications, more information on which appears in later sections of this report. We provide more details on our prototyping assumptions in the section on future work.

3.4 Application Global State and Recovery Scenarios

Global applications establish a view of the associated global state and proceed to work under that view until the application can detect or is informed that the view is inconsistent. The global state includes such information as how many applications participate and on which nodes, the set of resources contributed by the various applications instances (and the associated resource identifiers), and the consistency of shared data structures in GSM. The global state is assumed to be under possibly constant flux in parallel with application execution as processes, operating systems, and hardware resources join and leave the multicomputer or fail in ways not necessarily detected until the next access. This turmoil may also affect the associated API objects such as GSM sections and global mutexes.

Among the scenarios for dealing with fluctuations in global state are:

- Processes join or leave the global application. Shared data structures may be updated to reflect the new membership and algorithms may need to factor in the new set of participants.
- The process responsible for global application service on a particular node aborts and is restarted, contributing a new set of associated global resources. Remote references to the old set must be closed and switched to the new set.
- A process or operating system crashes during initialization, automatically cleaning up locally contributed global resources. This invalidates identifiers for those resources in the case where the identifier has been previously communicated to other processes through shared data structures but not yet accessed remotely. An attempted reference to such an identifier will fail.
- A global mutex is found not to have been properly released before termination of the process previously holding the mutex, indicating that the protected data structures may have been corrupted through a partial update.
- A GSM section is found to encompass failed physical memory or is no longer reachable.

Consider a thread executing a simple code sequence that looks up the identifier of a global mutex from a known GSM section that has been previously mapped, locks the mutex, updates a data structure in the GSM area, and releases the mutex. During the execution of this code the following application-visible disruptions in global state may occur:

- The mutex identifier may be found to be invalid (the contributing process or associated operating system died with no open references to the mutex held, so the object was reclaimed).
- The memory access to look up the mutex identifier in GSM may fail.
- An attempt to lock the mutex may timeout or may return the "abandoned" indication, such that current contents of the data structure cannot be trusted.
- The memory reads and writes to update the shared data structure may fail.
- The set of processes or operating systems involved in hosting and managing the GSM and/or mutex may change, requiring that a different set of global resources be used instead. A new global application process may wish to use a new mutex or GSM object in place of the old one, concurrent with execution of our example thread.
- Some other recovery event may be signaled by another thread, requiring the mutex be unlocked before global state may be fully rebuilt.
- The GSM containing the mutex identifier fails after the mutex is locked, complicating the task of unlocking the mutex if the process has not saved a local copy of the identifier.

Part of our task in supporting multicomputer applications is to ease the development and execution of applications in such a dynamic context.

3.5 A Recoverable Component Framework

The MCS Global APIs described previously provide primitives for global operations that correspond closely to local OS primitives for accessing local objects, such as mapping shared memory and waiting for synchronization objects. Many applications also need to perform other higher-level operations in a multicomputer context: initiating and detecting application-level recovery events, determining application-level cluster membership, and so forth.

We wished to facilitate multiple global "components" within one application, and to facilitate development of recoverable services for a variety of global applications, as an alternative to concentrating effort on making one particular application operate globally. For example, in a following section we'll discuss a few possible global components of Web servers, each of which can operate independently of the other components, and not all of these components would be appropriate for all Web servers. This led us to develop a model for encapsulating global components and a framework that could, for example, orchestrate the delivery of failure notifications to multiple components within a process and manage inter-component recovery dependencies.

Development of higher-level abstractions of multicomputer APIs is also prompted by the need to ease creation of multicomputer applications. Multicomputer applications require coding under special APIs, employ a programming model where global state periodically shifts underneath application threads, and must take the usual effects of distributed, concurrent execution into account. It is therefore difficult to produce multicomputer software, and any means to ease the burden of application developers is of potential value.

We created reusable code for such purposes in an abstract class named *McsComponent* from which multicomputer application components may be derived. The *McsComponent* class provides services for the derived component that include the following:

- A framework for initiating, detecting, synchronizing, and handling global system and component failures and recovery events, further discussed in the following sections. *McsComponent* detects the event, synchronizes with worker threads, and executes the recovery routines for the derived component after the underlying *McsComponent* infrastructure itself is made consistent with the new global state.
- Global component membership management, where all instances discover and agree on the set of instances participating in a generation of the global component. *McsComponent* joins a component group automatically at the time the derived component is initialized, and reevaluates membership upon a failure.
- Automatic creation and failure handling for certain common constructs, such as creation of a shared memory area for initial inter-component communication.
- Various library functions that help automate recoverable programming. Failure conditions are automatically checked and global component recovery is automatically triggered upon detection of failures triggered locally or remotely, in order to abandon current processing based on an inconsistent view of global state.

We add more detail on these topics in the sections that follow.

3.6 *McsComponent* Recoverable Programming Model

Each component instance maintains information that is sufficient to rebuild the local contribution to global state. Upon failure, the global state is rebuilt from scratch using contributions from each surviving component. Partial updates to shared data structures and other application-level effects left behind by failed instances are made consistent by the instance contributing the data structure.

We classify application threads into two types: “worker threads”, which perform the usual work of the component in response to requests from the MCS-aware application, and “recovery threads”, which initially establish a consistent view of global state and which reestablish that view in response to failures within that state.

3.6.1 Worker Thread Global State Access

McsComponent-derived code for a “worker thread” accessing global resources commonly follows the structure shown in Figure 3.

```
retry:

LockoutRecovery();              // Let current recovery finish, then lockout recovery

switch (AccessGlobalState())    // Access GSM, wait for a global mutex, etc.
{
case OK:                        // No error, continue
break;

case RECOVERING:                // Error; recovery is pending
ReleaseResources();             // Release local and global resources that conflict with recovery
AllowRecovery();                // Allow recovery to run
goto retry;                     // Wait for consistent global state and retry access

case UNRECOVERABLE_ERROR:       // Error; cannot recover
ReleaseResources();             // Clean up local and global state
AllowRecovery();                // Allow recovery to run
return UNRECOVERABLE_ERROR;
}
```

Figure 3: *McsComponent* worker thread recoverable programming model

We call attention to these details on the recoverable programming model for worker threads, some of which are illustrated in Figure 3:

- The code that detects failures is often a subroutine called by other code in charge of the higher-level operation that may need to be retried, often an *McsComponent* library function. We must also distinguish between errors that may be recovered from and those that may not. We chose to use a set of function return values to communicate this information between C language scopes.
- Most code must check not only for locally detected failures but also for remotely detected failures signaled through the global component recovery event, without distinguishing the two; they are handled alike.
- Recovery is always a global event, never simply a local occurrence. Each global component must have the same view of global state. Each component is informed of the event and then engages in appropriate recovery.
- Recovery runs in parallel with worker threads. The accessor makes a call to initiate a recovery event; recovery handling then proceeds for all components asynchronously with any concurrent worker access. Because recovery may be instigated by any component at any time, it must run as a separate thread of execution.
- Worker threads and recovery threads must synchronize with each other. This is required primarily because worker thread access relies on a consistent view of global state that cannot be changed until the worker indicates that it is safe to do so. In addition, worker threads may hold resources such that recovery would be inhibited from executing to completion, or such that the worker and recovery threads could deadlock with each other. This requires the worker to release resources and then get out of the way until recovery completes.

- The application code explicitly retries access upon failure. Held resources are released prior to starting recovery. The application code jumps back to the appropriate point for a retry of the higher-level operation being performed, which may actually involve several individual accesses to global state that require a consistent view (read the ID of a global mutex from a particular address in GSM, open it, wait for the lock…).

3.6.2 Recovery Thread Code

Since we handle multiple failures at a time, including failures during recovery itself, code for a recovery thread is also written along similar lines, with a couple of key differences.

Recovery thread code does not explicitly synchronize with other threads, neither worker threads nor recovery threads. *McsComponent* ensures only one recovery thread per node runs at a time, with any concurrent workers that wish to access global state blocked at *LockoutRecovery*() until recovery completes.

Recovery also does not retry failed operations in the sense shown above, since the entire recovery operation must either succeed or be retried from scratch. For example, failure to open a global object whose identifier was provided by a remote component might indicate a process or OS crash, which requires a reevaluation of global component membership, possibly re-electing a component master, cleanup of any data structures that apply to the failed component, etc. In other words, recovery does not rely on a consistent global state for higher-level operations – it establishes a consistent view of global state that must be reevaluated upon failure to establish such a view. A recovery thread that encounters a new failure relinquishes control to a new recovery thread that restarts recovery from scratch.

3.6.3 Initialization Code

We structure the derived component initialization code into two parts: one-time initialization and multiple-time recoverable shared state building.

One-time initialization must have no dependencies on non-local resources, such as depending on the correct operation of other components or on the availability of remote GSM. Initialization should not fail due to a situation that may be rectified with a recovery of shared state. (It is, however, quite difficult to ascertain that every possible error situation incurred by an operation intended to be purely local could not in fact be affected by global concerns.) If initialization of a component fails, it is as if the component never existed so far as the global state is concerned.

A successful initialization of shared state is also required before the component can do useful work. Any attempt at building shared state may run into a failed remote resource or may be interrupted by a global component failure event signaled by another component. Upon first-time initialization of the component, shared state builds are executed until one completes without incurring a recoverable failure. Shared state was either built successfully, indicating that the global component was in healthy shape moments ago and probably continues to be ready to provide service, or was built unrecoverably unsuccessfully, indicating the component cannot be ready for use due to problems other than recoverable global dependencies.

3.6.4 Modifying the Local Contribution to Global State

Both the local state from which global state is built and the global state must be modified to update the local contribution to global state. A common sequence for modifying global state is:

1. Modify the local information (which could be stored in local or global resources) from which global state will be built.
2. Modify the global state.
3. If step 2 finds that the information is already updated in global state then a recovery happened between steps 1 and 2, rebuilding the state for us from the local information we updated. Abandon the modification.

Portions of the processing for step 2 and perhaps also step 1 (if global resources are used for local state information) must lockout recovery and must abandon and retry certain operations upon a pending recovery, as usual for a worker thread.

We could require recovery to be locked out during all of steps 1 and 2, eliminating the need for step 3. Note that we cannot reverse the order of steps 2 and 1 unless we provide a way to check whether a recovery event occurred between the two points and redo global changes if so, since an intervening recovery would remove the global changes.

3.6.5 Automating Recoverable Programming

We are interested in pursuing the lengths to which realistic recoverable programming models can be provided by library functions or made implicit instead of explicitly coded. Our work to date has focused on a component framework and library routines where recovery is automated in the following ways:

- The appropriate failures that may be incurred by an operation are automatically detected, such as mutex abandonment, timeouts, and memory failures. Component termination and system-detected events, such as nodes that fail to deliver a heartbeat, are monitored constantly.
- Global component recovery is automatically initiated by the first component to detect a failure, and without the need for each component to explicitly start a recovery action.
- Operations automatically detect that recovery has been requested and abandon processing so that recovery may run, including interrupting certain blocking operations, such as waiting for global events and sleeping for a period of time.
- Recovery attempts are automatically serialized, and are synchronized with worker threads, in some cases through explicit calls by applications and in some cases automatically by library routines.
- An application-defined shared memory area is automatically allocated and relocated to surviving nodes upon failures. For some applications this may be the only shared data structure needed.
- Global component membership is automatically discovered. A single “component master” is automatically elected from that set, providing for component-defined actions

relegated to a single component per membership generation, which might include initializing certain shared data structures or signaling initialization events.

Further work we may pursue in automating recoverable programming is discussed in the section on Future Work.

4. Example Application: A Cooperative Web Cache

To demonstrate our MCS prototype, we modified an existing commercial application to use global resources and to recover from certain software failures. Our primary goal was to demonstrate kernel and application recovery from an OS crash on one node while shared resources were in use by the crashing node.

The MCS project was aimed at traditional scaling enterprise applications such as database managers and OLTP monitors. In fact, an earlier phase of the project demonstrated a global application version of the TimesTen in-memory database manager [2].

For the demonstration purposes of the most recent phase of the project, we wrote a Web server file cache that reduces disk I/O by caching local static files in memory (also known as a "reverse proxy"). We chose Web caching as an example of software for the commercial market that uses memory to advantage (many references, for example, [12, 13]) that we could quickly adapt for MCS purposes without proprietary source code and without dealing with the well-known rigors of modifying such software as database managers [10, 11]. This is not an ideal application for demonstration of multicomputer platforms, as the primary data being shared is read-only and easily replicated and partitioned across shared-nothing servers for scalability and availability with little or no inter-node communication (many references, for example, [9, 27]). The Web cache application does, nonetheless, share certain read/write fields and did allow us to investigate and demonstrate various aspects of the recovery model and the performance scalability characteristics of the platform. We also chose Web servers in general as a good vehicle for demonstration of various other planned MCS technology not presented here.

4.1 Design of the Web Cache

The Web cache is implemented in a Web-server-independent C++ class named *McsWebCache*, using the MCS APIs and *McsComponent* class described previously. Web-server-specific code adapts the *McsWebCache* object to the Apache Web server using the Apache dynamically loaded module interface, and to the Microsoft IIS Web server using the IIS ISAPI interface as a combination of an ISAPI filter and an ISAPI extension. Different nodes may run either Apache or IIS concurrently; all will share the same cache. Each node may run a Web server that participates in the global cache.

Figure 4 depicts the Web server software layers involved and the point at which MCS global resources are employed. The global Web cache resources include GSM sections and global mutexes for the cached file data, hash chain bucket heads and locks, and so forth. At an underlying level these also include the GSM, global mutexes, and global events used by the *McsComponent* object from which *McsWebCache* is derived. These resources are allocated and

contributed to the global application by the instances of the *McsWebCache* component running on various nodes in the multicomputer.

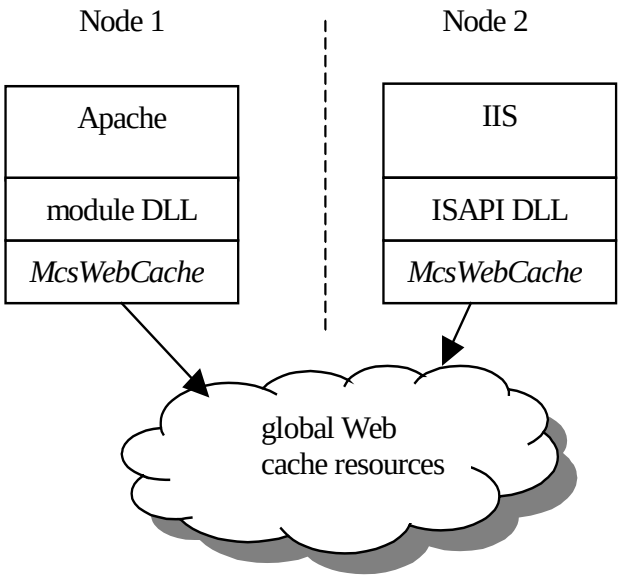

Figure 4: Global Web servers on two nodes accessing a common cache

Each *McsWebCache* component contributes an equal share of memory for caching files to the global pool. One component – the "component master" – also contributes memory for the so-called Master Table (automatically allocated by *McsComponent*) that contains hash table chain heads and other inter-server shared data structures, such as information on how to access each server's file cache memory area. All GSM contributed by a component is taken from GSM housed on the node where the component executes.

The hash table holds the heads of linked lists of cached files, according to a hash function applied to the file path string. Each hash chain has an associated global mutex to protect concurrent access. Each hash chain entry is linked through fields in a data structure co-located with the file data. A file appears only once in the cache; files are not replicated on multiple nodes for locality, as the MCS architecture is intended to maximize the benefit of utilizing global resources versus locality-aware resource duplication and programming models. The replacement policy is LRU. To simplify memory management and access to the file data, each file is cached in contiguous virtual memory within one of the memory areas (see Figure 5). Each server manages the memory of its own contribution only, adding new files and evicting previously cached files as needed from the GSM it contributes. This can lead to some imbalance of memory utilization, but saves considerable overhead that would be incurred by global memory management.

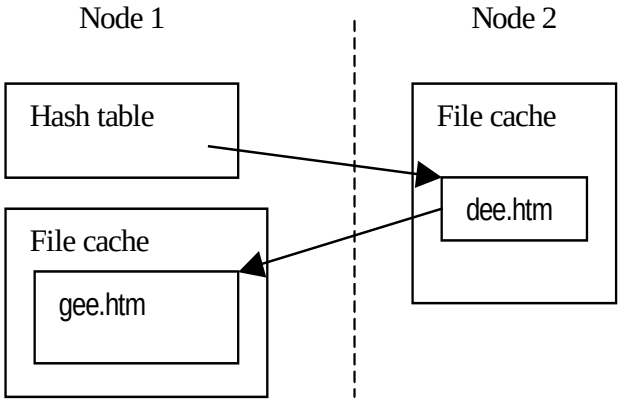

Figure 5: Two files on a hash chain over two nodes

4.2 Recovery from Failures and Changes in Global State

Each Web cache object instance keeps separate information about its own important contributions that is sufficient to rebuild the local contribution to global state. Upon failure or change in membership of the global application, the global state is rebuilt from scratch using contributions from each surviving component. Global state contributions include the identifier of the GSM area for the local cache contribution (and any cached data therein), the hash chain entries for locally cached files (a separate list of these is maintained separately by each component), and, for the component master, the identifiers of the global mutexes that protect the hash chains.

This application can recover from multiple failures at a time. This is possible in part because the information contributed by a node in GSM may be safely discarded, as the information is persistent on disk. It is therefore not necessary to provision more memory on the remaining nodes upon recovery to subsume a failing node's contribution.

Failure and reconfiguration recovery scenarios for *McsWebCache* code include:

- Detection of an abandoned hash chain mutex, indicating failure of a process (or OS) while updating data structures such as hash chain pointers and cached file data reference counts, possibly leaving these in an inconsistent state. The hash chain must be rebuilt and references from failed processes cleared. Reference counts are implemented per-instance for this purpose.
- Processes enter and leave the global application concurrent with ongoing cache access or recovery, requiring coordinated changes in the global memory sections and perhaps mutexes in use. Current access to resources to be deallocated is allowed to drain out and then references are dropped; new references are made to new resources. If opening a new GSM section or mutex fails then the contributing process/OS failed prior to the first remote reference, and membership must be reevaluated. If the "component master" fails, a new master is elected (by *McsComponent*) to recover the hash table and coordinate

recovery activities. If processes have left the global application then the associated cached files are removed from the hash table.

- GSM access may fail while building shared state, while traversing hash chains and updating cached file data reference counts and access times, and while sending cached data back to the HTTP client.

Figure 6 shows an example code fragment that might be used by *McsWebCache* to retrieve a pointer to the first cached file on the hash chain given by variable *index*. Code to actually access the cached file and clean up afterwards (unlock mutex and re-allow recovery) is omitted for brevity.

```
retry:                                  // Here upon each try at access to consistent global state

LockoutRecovery();                      // Lockout recovery, begin sequence that requires consistent global state
pHead = LocalBucketInfo[index].pHead;   // For clarity, copy the hash bucket head pointer and the global mutex
MutexId = LocalBucketInfo[index].MutexId; // ID protecting the chain to simple local variables

switch (McsWaitForMutex(MutexId))       // Lock global mutex protecting hash chain
{
case MCS_OK:                            // No problems detected, continue
break;

case MCS_RECOVERING:                    // Recovery is pending
AllowRecovery();                        // Allow recovery to run
goto retry;                             // Try again

case MCS_UNRECOVERABLE_ERROR:           // An error from which we cannot recover
AllowRecovery();                        // Allow future recovery to run
return MCS_UNRECOVERABLE_ERROR;
}

pCachedFile = *pHead;                   // Read the GSM containing the hash chain head

if (DetectMemoryFailure())              // Memory failed somewhere, perhaps the read of chain head failed?
{
  McsReleaseMutex(MutexId);             // Release resources conflicting with recovery: unlock hash chain
  AllowRecovery();                      // Allow recovery to run
  goto retry;                           // Retry access
}
```

Figure 6: *McsWebCache* worker thread code example

The *LocalBucketInfo* data structure is a local copy of hash table information created by the recovery thread. This exists primarily to ensure that a memory failure cannot obscure a mutex ID that must be unlocked for recovery to occur, as we will discuss later.

We have demonstrated that the application recovers from multiple process failures at a time in each unique situation where the failure could affect other processes. The MCS project has not yet reached the point where the complete system recovers from an OS crash, but several software components of the system have individually demonstrated this under prototype conditions. The Web cache application should be able to recover from a remote OS crash in exactly the same fashion as process terminations are handled; recovery from OS crashes is a goal we are working toward. The prototype platform is not suitable for demonstrating recovery from hardware failures, as the interconnect cannot recover from unresponsive nodes.

4.3 Impacts of Recoverable Programming

Apart from structuring the code as previously described for *McsComponent*-derived components, further samples of significant effects of recoverability upon the Web cache code are presented in the sections that follow.

4.3.1 Crossing Multicomputer Awareness Boundaries

The "multicomputer-aware" application ultimately uses global resources to perform application-level work. If the application hasn't been written to be entirely multicomputer-aware, or if part of that work involves system calls or library routines provided by a non-multicomputer-aware commodity OS, then the designer must wrestle with how to cross the multicomputer-aware/multicomputer-unaware boundaries.

The Web cache code allows the existing, non-multicomputer-aware Web server to access file data in GSM when sending data back over the network to a client, using existing functions of IIS and Apache. The Web cache code itself also uses Win32 I/O routines to read file data into GSM. In both cases, we clearly cannot take the performance hit of copying massive amounts of file data between GSM and local memory (that "fails with the node/OS/process") in order to prevent passing a reference to global resources outside the sphere of our multicomputer-aware code. We must therefore use multicomputer-unaware code for these purposes and must consider how failures will affect that code.

We take OS/process crashes into account when passing pointers to file data in GSM to Web server routines that send the data back to the client. One of our chores upon OS/process failure is to unmap the GSM contributed by that node to the cache. If that section is unmapped while a worker thread is referring to an address in that section then the transfer is disrupted with an exception. We could avoid this disruption by delaying unmapping the section until all current requests using the section drain out. That would come at the expense of dealing with the problem of restarting a new Web cache component on the node for which not all previous resources have yet been released.

Proposed memory failure recovery mechanisms that satisfy failed reads with "dummy data" pose some problems for unaware code. One example is that cached file data in our GSM buffer will be copied by MCS-unaware networking routines into local memory before DMA to the networking interface occurs. If a memory failure occurs during the copy, incorrect data will be copied. If that data is transmitted to the HTTP client then the failure is propagated off the

multicomputer system, exhibiting poor fault containment. Future work may concentrate on catching the problem in MCS networking drivers.

Another example is the use of null-terminated strings in GSM. Failure of some or all of the memory hosting the string would present what looks like a string of dummy characters, possibly without the proper trailing `NUL` byte, artificially extending the length of the "string". The dummy data may even fill all pages of the GSM attachment in the virtual address space of the process. A non-MCS-aware *strcpy()* on a source string in GSM might, for example, copy merrily away until incurring a memory exception on an invalid source or destination address, perhaps overwriting other random data in the address space beyond the intended destination string. This is a case where the failure alters behavior of the MCS-unaware system routine in such a manner that the failure is propagated before the MCS-aware caller can detect and recover from the failure. The Web cache code is careful to use *strncpy*() and other such bounded forms of string calls, assuming that valid information on the intended length or maximum length of the string is available (perhaps implicit or perhaps successfully read from GSM), and that the memory mappings for that many bytes continue to be valid.

4.3.2 Avoiding Reliance on GSM for Recovery Instructions

Awareness of memory as something that can fail without concomitant failure of the accessing code can lead to changes in the design beyond basic checking for access failures.

For example, at first glance it would seem appropriate for Web cache access routines to generate a pointer to certain data structures in global memory and pass the pointer to various functions that manipulate the data structure. One such data structure is a hash chain head, which includes the identifier of a mutex that protects the associated hash chain. Yet, if the application were to successfully lock the mutex via dereferencing the identifier through that pointer, and then memory were to subsequently fail prior to unlocking the mutex, the application would have lost its record of the mutex that must be unlocked. The Web cache code copies such information to local memory that "fails with the process" prior to use, so that memory failures can be survived without relying on the failed memory to instruct how to recover.

4.4 Performance Results

Although we did not perform extensive performance tuning of the Web cache code, we did characterize certain aspects of performance using the Apache Web server and a workload based on a popular Web server benchmark that serves static content.

Of primary interest for the current phase of the project is whether applications quickly recover from failures and resume service. For our Web cache demonstration workload, application-level recovery from failure occurs in less than 2 seconds, during which time the remaining processes discover the new membership, rebuild shared data structures, clean up resources held by the failing process(es), and then continue service. MCS kernel-level recovery from node failure has been partially evaluated. Recovery of the GSM manager and supporting components requires approximately 25*ms* for a system with 64 application GSM areas allocated (plus 8 taken by MCS system components) [2]. In order to better understand how these results compare to other availability solutions we may examine figures for failover high availability products. Reported

application recovery times include 15 to 30 seconds for Compaq TruCluster Server V5.0, touted as one of the fastest failover rates in a Giga Information Group report [18] and about 30 to 90 seconds for Microsoft Cluster Server on NT 4.0 [19]. Failover products perform substantially different functions than that of our recoverable applications, and so a direct performance comparison may be misleading. We can, however, gain some idea of the application recovery times prevalent today and can generally assert that the shared memory architecture provides a high performance availability solution for our application.

Figure 7 shows benchmark results for configurations that omit the Web cache, and for configurations that substitute local Win32 resources for MCS global resources on a single node, normalized to the results for "everything local".

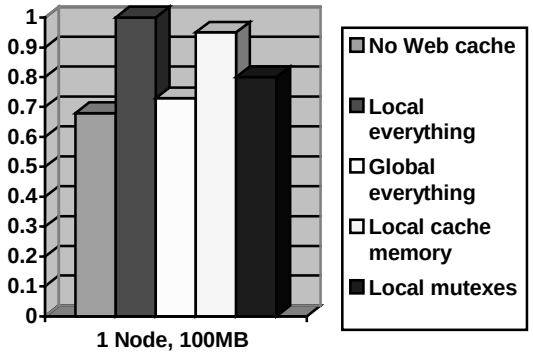

Figure 7: Local vs. Global Resource Performance

The first bar shows the performance of the Web server without the *McsWebCache* component in an otherwise identical configuration to the other measurements. The second bar shows the Web server executing the *McsWebCache* component with the full recoverable programming model (locking out recovery threads and so forth) but using Win32 local memory and mutex resources instead of MCS global resources. At a 47% improvement over no caching, we can conclude that this Web server greatly benefits from memory caching. More importantly, we can also conclude that the penalty paid for recoverable algorithms is not exorbitant compared to the savings in disk access that would be incurred without caching – when global resources perform comparably with local resources.

Yet the full MCS version with global memory and mutexes, shown in the third bar, shows only an 8% improvement over no caching. The primary reason for this is illustrated by the rightmost bars, which show performance of versions that use local resources for one or the other resource in use during a benchmark run: one version that uses local memory for the cache but continues to use global mutexes (including some use of GSM for global mutexes), exhibiting an improvement of 40%, and one that uses local mutexes and global memory, a 17% improvement. The MCS prototype platform incurs an access penalty to GSM, local or remote, of approximately 12X [2] – decidedly not an ideal platform for our locality-unaware programming model. The Web cache application is paying a heavy price for the access penalty, even in a single node configuration. Figure 8 shows the flat scaling of additional global memory on a single node compared with the

beginning of a more linear scaling of local memory, normalized to the results for 100MB of local memory.

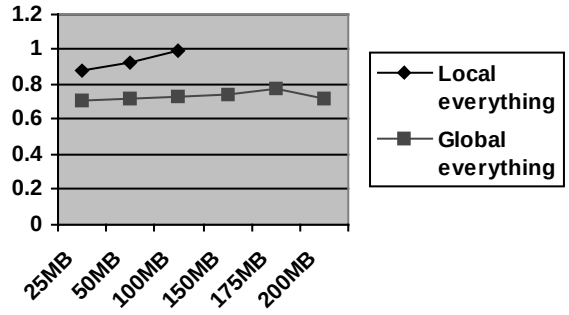

Figure 8: Scaling of local vs. global memory

Figure 9 shows results with varying sizes of cache memory per node for one and two node configurations (three and four node results are unavailable at this writing), normalized to the results for a single node at an optimal 175MB memory.

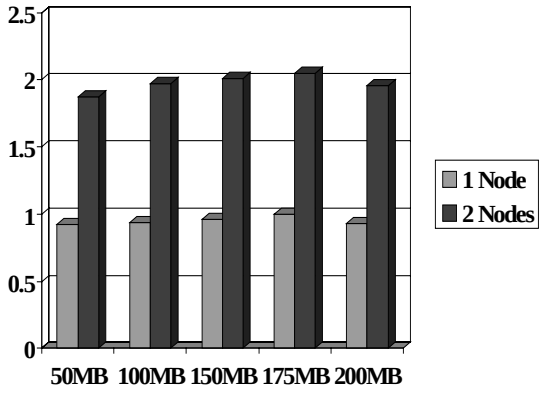

Figure 9: Benchmark results for 1 and 2 nodes

We consistently obtained 2X performance when resources were doubled by moving from one to two nodes with double the amount of aggregate memory for the cache. We also achieved 2X scaling when moving from a single node to a two node configuration where the aggregate size of memory is equivalent but split across the two nodes, indicating that memory size is not our scaling bottleneck. We can expect to obtain better than 2X scaling when memory is doubled on a suitable platform because the effective cache size of a shared nothing configuration is not doubled due to duplication of frequently accessed content in both caches. In the above trials we saw a maximum of 11% scaling beyond 2X, with an average of 7.4%, an insignificant amount. The prototype platform is not suitable for demonstrating applications not written to be aware of memory placement, and we cannot demonstrate a performance advantage for multicomputers over conventional clusters using this application and this platform.

5. Perspective of ISVs and Partners

In addition to researching technology issues related to platform software and hardware, the MCS program examined business issues related to the acceptance of such a platform in the commercial marketplace. HP consulted with researchers and developers at a variety of potential business partners, including leading operating system and commercial application vendors. The comments that follow do not represent the official positions of any particular company, but indicate perspectives with which major ISVs view shared memory multicomputer platforms.

5.1 ISV Concerns

The first major concern voiced by the ISVs was the **standardization of the global APIs**. This is an understandable concern, because maintenance of a source code tree and testing of code functionality are complicated by variations in APIs from one platform to the next. There are at least three levels of standardization relevant here: the first is that the vendor of the host operating system certify the global APIs. Since global APIs affect such fundamental matters as process scheduling and virtual memory management, it is important to guarantee that they are well integrated with the host operating system. For example, in the case of MCS, which was based on Microsoft's Windows NT, the ISVs would prefer that Microsoft endorse the MCS global APIs and support them through future updates of the operating system. The next level is that all shared memory multicomputers based on Windows NT should share the same global APIs. The final level is that all shared memory multicomputers, even those based on other operating systems, should share some similarity in API structure, to facilitate porting from one platform to the next. This concern is not unique to shared memory multicomputers; every new interface faces the challenge of establishing a standard so that the virtuous circle of applications and platforms feeding off each other can be initiated.

The second major concern voiced by the ISVs was **the approach taken to achieve scalability** across multiple nodes, which may be at odds with their existing strategy. ISVs who have committed to a shared nothing model, for instance, may be reluctant to embrace the shared memory multicomputer model.

The last major concern relates to the added **difficulty of achieving high availability** under the MCS failure model. A shared memory multicomputer can be used to create highly available applications, but it does introduce new failure modes not shared by other computing platforms. If data stored in global memory is corrupted or if the hardware supporting global memory experiences a failure, the consequences can directly affect every node sharing that memory. This requires the multicomputer applications programmer to exercise rigorous discipline to ensure that any global memory failure can be compensated for (for example, by maintaining redundant copies of global data structures). This discipline is not required of programmers for other platforms: on an SMP, a memory failure generally causes the entire application (sometimes the entire system) to crash; on a shared nothing cluster, it is assumed that the failure of any one node has no consequences for any other node.

5.2 Example Application: A Database Manager

Here we describe in some further detail our consultations two years ago with a company regarding a previous version of their database manager product, in order to shed light on its suitability as an MCS application and the issues involved in considering a port to MCS.

The product is structured as a set of cooperating server processes plus a set of "background processes". The server processes perform the actual work of executing database queries and updates. The number of server processes is tunable, usually chosen to be a small multiple of the number of CPUs in the system. The background processes perform a range of housekeeping tasks on behalf of the entire database manager, rather than on behalf of a particular user or query. The processes share a pool of memory of up to several gigabytes that contains a wide variety of data structures. An important use of the shared memory area is to cache file blocks of the database, since a significant performance advantage can be realized if the index blocks of a frequently used table are accessed from memory rather than from disk. For updates, the dirty file blocks can be buffered in the shared memory area while waiting to be copied out to disk.

On an SMP, the product can achieve high throughput by executing multiple queries in parallel. The data structures in the shared memory area were carefully designed so that locks can be used to enforce the necessary mutual exclusion, while permitting simultaneous access to independent data items. This suggests that the product would be a natural application to globalize on an MCS system. The shared memory area could be placed in global shared memory. Each node of the MCS would support a set of server processes, which are already capable of sharing the global data structures in a consistent manner. One would expect that the throughput of the global application would scale well with the number of MCS nodes.

Dealing with the background processes would be more complicated, since the product currently expects exactly one background process of each type. Concurrent access is not expected, and therefore each background process does not lock its own data structures. It would require a significant amount of work to globalize each of the background processes so that each MCS node could have its own set without corrupting the data structures that would be shared among the processes. The alternative of keeping exactly one set of background processes for all nodes would be an availability issue if the background process were to fail.

The database manager is an extremely robust product, from the standpoint that it maintains data integrity in the face of failures of the server and background processes and in the event of a system crash. Before a server process updates any global data structure, it saves away the previous state. If the server process fails before being able to take the global data structure to the next consistent state, the background process responsible for global data structure repair will notice this fact and restore the structure to its previous state using the information stored previously. All database updates are also written to a log file on disk. In the event of a system crash, the log files are replayed when the system recovers in order to return the database to a consistent state. Rebooting a crashed system and replaying the log files can take many minutes, during which time the database is unavailable. An MCS version of the product would be required to match the availability characteristics of the existing version in order to be valuable, requiring the suggested changes to globalize the background processes. This would actually increase the availability of the system, because the current product is forced to reboot if any of the background processes fail.

If the shared memory area were placed in global shared memory, it would be vulnerable to the failure of that memory. It would certainly be a major undertaking to rewrite the product, a large base of carefully tuned code, to incorporate the discipline necessary to protect against memory failures.

In considering whether to adapt the product for MCS, developers and management must consider that the effort required could instead be spent tuning the performance of their existing clustered offering, which has a different set of performance and availability trade-offs compared to the MCS approach.

5.3 Addressing ISV Concerns

We began work on the *McsComponent* recoverable component framework described earlier in order to help address the difficulties of attracting ISVs to multicomputer platforms.

We also planned for a developers kit, to include various global status display and modification tools as MCS counterparts to the tools shipped with the NT Resource Kit and Platform SDK. The developers kit was also to include application and kernel driver debugging tools adapted for concurrent, distributed debugging, which remains a weakness with most development environments [15, 16]. We planned to leverage ongoing work by the Parallel Tools Consortium High Performance Debugging Forum [17] to standardize distributed multiprocess debugging, with such features as broadcasting commands to sets of threads in multiple processes and aggregating output from multiple threads. NT kernel debugging currently may be performed remotely over serial cables, the setup of which does not scale to large numbers of nodes; we investigated translating the serial protocol to a TCP protocol over LAN, hooked into an intelligent I/O processor on the target and using shared memory to distribute the debugger traffic to the proper nodes. And there is a fault containment angle to the debugging tools: debugging a kernel must not cause the other nodes to eject the node from the cluster due to missed heartbeats while kernel execution is frozen (exhibiting the symptoms of a fault), as occurs today.

6. Related Work

An example of a commercially available NT-based shared memory multinode platform is the Unisys Cellular MultiProcessing (CMP) architecture [3]. Global memory is used as a fast communication channel as an alternative to a LAN; the resource is not exposed directly to applications. Application programming models, scalability, and availability characteristics are therefore essentially the same as a shared nothing cluster with a fast interconnect. Application sharing of data in shared memory is mentioned as a possible future enhancement, at which time impacts of software failures upon high availability of the entire system will become an issue.

Also available in the industry is the Compaq OpenVMS Alpha Galaxy platform [4], which executes multiple operating systems in different partitions interconnected through shared memory. APIs are provided for application access to such global resources as memory, events, locks, and even processors, which may be dynamically assigned to nodes. Fault containment and recovery for global applications are not addressed in the literature available.

Sun Microsystems Laboratories investigated multicomputer systems in the Solaris MC project [28], which was primarily concerned with presenting a "single system image" over a cluster of LAN-connected systems, each running a separate OS. Unmodified applications can make use of global resources such as file systems, I/O devices, network connections, and executing processes remotely. Published reports provide only a little insight into the fault containment characteristics of the architecture, which has fault containment at node granularity as a goal, but any processes using resources on a remote failed node are probably aborted.

Academia research into fault containment for shared memory multinode computers was performed at Stanford in such projects as Hive [7] and Cellular Disco [8]. These architectures divide nodes and subsets of SMP hardware into fault containment units called *cells*, such that failure of part of an SMP, such as failure of one processor, only brings down operating systems and applications that depend on the associated cell, providing for fault containment within an SMP, which is not addressed by MCS.

Hive comes closest to the goals of MCS, allowing OS software to survive access to failed GSM using a "careful reference" sequence and allowing applications to survive GSM failures in situations where no possibility of data inconsistency exists. Hive also targets misbehaving software that corrupts GSM data structures through "wild writes", which is not addressed in MCS. Hive does allow data corruption to slip by undetected in certain situations and does kill processes that it determines *could* experience data corruption, leading to their useful definition of application fault containment:

> *A system provides fault containment if the probability that an application fails is proportional to the amount of resources used by that application, not to the total amount of resources in the system.*

…although the possibility of operating system failures makes the situation less clear. Similar goals for MCS could be stated as:

> *A multicomputer provides fault containment if the probability that an application fails is not increased through the use of resources provided by remote nodes.*

Cellular Disco concentrates on containment of hardware failures, without targeting failed access to shared memory. Containment occurs at a larger granularity than MCS, facilitating partial failures of virtual operating systems rather than individual processes by rebooting virtual machines using resources on a failing cell, with the attendant impacts on availability of the hosted applications. Failures such as node power loss and interconnect failure on a node were simulated, causing a crash of that node and reboot of the virtual machines and associated operating systems and applications holding references to resources on that node. Each application was structured to hold references to exactly two remote nodes, such that exactly two virtual machines were rebooted.

Both the Hive and Cellular Disco projects demonstrated fault containment under simulation using an unmodified parallel compilation application that distributes computation to a static set of processes in a "scientific clustered programming" model using remote memory. It is not clear quite how remote memory was used and to what levels inter-process communication or other interdependencies were present. In the case of Cellular Disco, the application was structured to limit its dependencies to a subset of nodes, such that a single failure would not kill all processes in the global application. MCS, by contrast, targets commercial applications that use large numbers of a variety of global resources across all nodes, and that employ dynamic process membership in the global application, for scalability and availability purposes rather than distributing computation.

A number of research projects have focused on fault tolerance for Distributed Shared Memory systems (which present a single address space abstraction over physically distinct memories) using techniques such as checkpointing and memory access logging to restore memory to the last known consistent state previous to a failure [23, 24]. These projects are normally prototyped on network-connected clusters, often using a locality-aware distributed programming model due to the algorithms and interconnects by which memory pages are shipped around. Such methods could be applied to shared memory multicomputers to solve the problem of keeping GSM data structures consistent, although many schemes are expected to have non-trivial performance impacts. General application failure recovery via transactions with checkpoint and restart has been the subject of a vast number of papers, some of which mention recovery of shared memory communication, but often with unclear applicability due to such concerns as performance (such as [25]). Memory areas insulated against software corruption have been used to speed performance of writes to permanent storage in transactional systems, but not as a communication means between processes, in such projects as Rio Vista [26], which is not an area targeted by MCS at this time.

7. Future Work

The following sections describe possible future directions for our work in multicomputer applications.

7.1 Declarative Specification of Transactions within a Global State

One could envision the property of "depends upon a consistent global state" being largely a declarative specification rather than a programmatic one, much like the property "transactional" in Enterprise Java Beans or the Microsoft Component Object Model. We are interested in pursuing a model where a series of operations that rely upon a consistent global state, such as locking a particular global mutex, accessing certain GSM areas, and unlocking the mutex, are begun as a "transaction" that is backed out and restarted upon failure. The component infrastructure transparently handles many of the details of backing out a transaction to the appropriate point, reestablishing a consistent global state, and restarting the transaction under a new global context.

7.2 Recovery from Memory Failures in Unknown Context

Methods for application notification of memory failures have been an area of ongoing discussion, such as comparing the merits of delivering exceptions or signals versus explicitly "polling" for previous failures. Under either method, applications check whether previous accesses have failed at appropriate times, generally after access to GSM and before continued execution under the assumption the previous access was successful would cause erroneous behavior.

Recovery from memory failures on present-day commodity processors poses some thorny problems in regard to notifying the proper application contexts of failed memory access. On IA-64, for example, these problems include imprecision of notification, speculative data prefetches, and advanced instruction retirement prior to completion of memory operations [1]. For our prototyping work, we proposed extensions to the MCS designs that allow for detection of memory failures that avoid some of these problems by allowing for recovery from memory failures in unknown contexts. The system is not required to match the failed access with the context that issued the request, placing a greater burden upon applications to detect failures and to recover from all failures anywhere in the system. The application may perform unnecessary repair work, but failures should be rare.

We propose the following additional properties of the platform, implemented in hardware or by the operating system:

1. The system keeps a count of the number of memory failures detected since power on. The OS could maintain this count in response to recoverable hardware notifications or this could be a hardware register of the platform. This count is global across all nodes.
2. A facility is provided for application software to inquire whether memory that it has mapped remains accessible. This facility can be restricted to operating on the local GSM contribution.

Software checks for an indication of "a new memory failure has been detected" at appropriate times under either an exception/signal model or a polling model. Software is never told whether any of its own previous memory accesses have failed, but is told only that a failed access was detected due to some operation. Software should then initiate a full recovery of all possibly

affected data structures to ensure consistency of that application. This checking is required at both the kernel and application level.

Software can inquire whether any of its virtual address space translates to physical memory that is now inaccessible in order to cleanup any references to newly unavailable data structures. This capability can be restricted to inquiries for the local GSM contribution. Unavailability of remote GSM mapped by the local node may be signaled by other means, such as notification of whole node failure (if appropriate) or modification of global state contributed by the remote application instance that detected the unavailability. Modification of global state might entail, for example, communicating a new identifier for a new GSM section allocated from the remaining GSM on the remote node.

7.3 Multicomputer Applications in the Internet Data Center

Ideas that we are considering for future MCS applications within the realm of Web server farms include:

- Global access to per-session data, such as "shopping carts", held internally by Web servers. An example is the Microsoft IIS *Session* object, which is used by Web applications (using interfaces such as Active Server Pages or Information Server API) to associate HTML cookies with session-specific data held in the Web server.
- Global access to Secure Sockets Layer (SSL) keys calculated by a Web server. SSL keys are expensive to compute.

Both of the above are aimed at avoiding server affinity and the attendant potential "hot spots" of imbalanced server loads, since load balancing of multiple HTTP requests referencing the associated local server resources is generally inhibited today. The associated service also gains higher availability through being available on all nodes. For example, a shopping cart implemented using an IIS Session object today will be lost if the associated Web server crashes. Other solutions could, of course, be employed to making this information available on multiple servers, such as to write a protocol for passing the information in messages in a distributed system model or to save the information in a globally accessible database. We may study the advantages and disadvantages of multicomputers in these contexts.

Other Web application technology that we are considering to demonstrate the use of global server resources include caching of database data and presentation objects, EJB or COM object activation pools, and various load balancing information.

8. Conclusions

We have discussed one model for writing shared memory multicomputer applications that quickly resume service in the event of a number of failure classes, and have used that model to build a working example of a commercial application. The application demonstrates certain advantages of multicomputers for writing highly available clustered applications: by rapidly reconstructing global state using low-latency access to global memory in an SMP-like programming model, applications may recover from faults using strategies that would be

impractical on other architectures. The application also serves as an example of how arduous a task it is to adapt software for fault containment on multicomputers, a number of aspects of which we've examined in detail. We demonstrated 2X performance scalability for two nodes with an untuned application in a domain not ideal for multicomputing, despite the overhead of maintaining redundant data structures, checking for failure events, and synchronizing worker and recovery actions. Better performance scalability is suggested for future platforms with better memory access characteristics.

The difficulties in adapting shared memory applications for fault containment are numerous, and there are significant hurdles to acceptance of multicomputer platforms by application vendors. There is little incentive to tackle the difficult problems of programming in a multicomputer environment without a clear customer base. The difficulties of recoverable programming paradigms can be mitigated through the use of frameworks that automate many of the chores of recoverable programming, an area deserving of further study. Standardization of multicomputer interfaces and provision of development tools adapted for multicomputer environments might also alleviate the situation.

Acknowledgements

We gratefully acknowledge Dejan Milojicic and Alan Messer of HP Labs and the program committee of the USENIX First Workshop on Industrial Experiences with System Software (WIESS) for reviewing and making suggestions on improvements to this paper.

References

[1] Dejan Milojicic, Alan Messer, James Shau, Guangrui Fu, and Alberto Munoz, "Increasing Relevance of Memory Hardware Failures / A Case for Recoverable Programming Models", to appear in the *9th ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, Kolding, Denmark, September, 2000.

[2] Dejan Milojicic, Steve Hoyle, Alan Messer, Albert Munoz, Lance Russell, Tom Wylegala, Vivekanand Vellanki, and Stephen Childs, "Global Memory Management for a Multi Computer System", *Proceedings of the 4th USENIX Windows Systems Symposium,* August, 2000.

[3] Unisys, "Cellular MultiProcessing and Uniform Memory Access", downloaded June, 2000.

[4] Compaq, "OpenVMS Galaxy Guide", May 17, 1999.

[5] Sharon E. Perl, Richard L. Sites, "Studies of Windows NT Performance using Dynamic Execution Traces", 1996.

[6] Pfister, Gregory F., "In Search of Clusters" second ed., Prentice Hall, 1998.

[7] John Chapin, Mendel Rosenblum, et al., "Hive: Fault Containment for Shared-Memory Multiprocessors", *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[8] Kinshuk Govil, Dan Teodosiu, et al., "Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors", *Proceedings of 17th Symposium on Operating Systems Principles*, 1999.

[9] Ludmila Cherkasova, "FLEX: Design and Management Strategy for Scalable Web Hosting Service", HP Labs Technical Report HPL-1999-64.

[10]Luiz Andre Barroso, Kourosh Gharachorloo, Edouard Bugnio, "Memory System Characterization of Commercial Workloads", *Proceedings of the 25th International Symposium on Computer Architecture*, June, 1998.

[11]Brad Day, "Trends in Cluster Architectures", Giga Information Group Thematic Planning Assumption T-1298-006, December, 1998.

[12]Martin F. Arlitt, Carey L. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications", *IEEE/ACM Transactions on Networking*, Vol. 5 No. 5, October, 1997.

[13]James C. Hu, Sumedh Mungee, Douglas C. Schmidt, "Principles for Developing and Measuring High Performance Web Servers over ATM", Washington University Dept. of Computer Science Technical Report WUCS-97-09, 1997.

[14]Krishnan Venkitakrishnan, "Investigation and Analysis of a Modular, Scalable and Fault Tolerant System Interconnect for Multi-Computer Systems", HP Labs Technical Report HPL-1999-152, 1999.

[15]Jonathan B. Rosenberg, *How Debuggers Work*, Wiley Computer Publishing, 1996.

[16]Ashis Tarafdar, Vijay K. Garg, "Debugging in a Distributed World: Observation and Control", *Proceedings of the IEEE Workshop on Application-Specific Software Engineering and Technology*, 1998.

[17]Parallel Tools Consortium web page at http://www.ptools.org/ .

[18]Brad Day, "Compaq's TruCluster Server V5.0 --- Single-System Advantage and Scorecard Overall", Giga Information Group *IdeaByte*, December, 1999.

[19]Charles Bruno and Greg Kilmartin, "Bulletproofing NT", *Network World*, June, 1998.

[20]David A. Patterson and John L. Hennessy, *Computer Organization and Design*, second edition, Morgan Kaufman, 1998.

[21]David L. Black, Avadis Tevanian, Jr., et. al., "Locking and Reference Counting in the Mach Kernel", *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[22]John H. Hartman and John K. Ousterhout, "Performance Measurements of a Multiprocessor Sprite Kernel", *Proceedings of the USENIX Conference*, 1990.

[23]Galen C. Hunt and Michael L. Scott, "Using Peer Support to Reduce Fault-Tolerant Overhead in Distributed Shared Memories", University of Rochester Computer Science Department TR 626, June 1996.

[24]Golden G. Richard III and Mukesh Singhal, "Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory", Ohio State University Computer and Information Science Research Center, March, 1993.

[25]David Ellis Lowell, *Theory and Practice of Failure Transparency*, PhD Thesis, University of Michigan, 1999.

[26]David E. Lowell and Peter M. Chen, "Free Transactions with Rio Vista", *Proceedings of the 1997 Symposium on Operating Systems Principles (SOSP)*, October, 1997.

[27]Mohit Aron, Darren Sanders, et. al., "Scalable Content-Aware Request Distribution in Cluster-Based Network Servers", *Proceedings of the USENIX 2000 Annual Technical Conference*, June, 2000.

[28]Yousef A. Khalidi, Jose M. Bernabeu, et. al., "Solaris MC: A Multi Computer OS", *Proceedings of the USENIX Annual technical Conference*, 1996.