Operating System Support for Shared Memory Clusters

Ronald L. Rockhold[‡†] James L. Peterson[‡]

‡ IBM Corporation. Advanced Workstations and Systems, Austin, TX
† Florida Institute of Technology. Melbourne, FL

Abstract

This paper addresses a purely software-based solution to the multiprocessor cache coherence problem by structuring an operating system to provide for the coherence of its own data while exporting coherent memory to user processes.

Also covered are the results of a proof-of-concept port of Mach 3.0, using the principles in this paper, to a prototype of the IBM Shared Memory System POWER/4™, a Shared Memory Cluster. This is believed to be the first implementation of a commercial operating system on a non-cache coherent machine and required the development of a software technique to detect coherence violations. Benchmark results show that on the four CPU system this solution provides a throughput increase of up to 3.9 times that of a single processor.

1. Introduction

1.1 The Cache Coherence Problem

The use of private caches in a multiprocessor (MP) creates the *cache coherence* (CC) problem. When a processor first references a data item, the *memory block* (typically 16 to 256 bytes) that contains it is loaded into its cache. Subsequent references to data items are satisfied from the cached copy. The CC problem occurs when two or more processors, using their private caches, share changeable data. A *data staleness problem* results when a processor's access to shared data is satisfied from a cached copy of a memory block which has been modified by other processors since it was last loaded. A *data integrity problem* occurs (in copy-back schemes where the entire memory block is copied back to memory) when two processors modify different variables that reside in the same memory block; regardless of the order in which the cached blocks are copied back to real memory, the correct value of both variables will not be reflected.

Solutions to the CC problem are categorized by the memory models they support. These describe how, when, and for which shared data items memory accesses are synchronized.

The strongest model, and the one most commonly and naturally assumed by programmers, is the one provided by uniprocessor systems. The execution of instructions in cooperating processes are interleaved -- memory changes caused by one instruction are immediately and consistently "seen" by all others. Multiprocessors which preserve this uniprocessor image of memory access coordination are defined by Lamport [Lam79] to be *sequentially consistent* and conforms to the *strongly ordered* memory access model. Until recently this was the only model that was provided by Symmetric Multiprocessors (SMPs).

The weakest model, which we will call *unordered*, provides no access synchronization between multiple processors that access shared memory.

Although there are other models that fall between these such as the *weakly-ordered model* [DSB86] (used in the Convex SPP) and the *release consistency model* [Gha+90] (used in the Stanford DASH system [Len+90]), this paper deals only with the two extremes where the operating system is adapted for execution on an unordered multiprocessor while providing a strongly ordered memory model to users.

1.2 Shared Memory Clusters

The *Shared Memory Cluster* (SMC) is emerging as an architectural base for Massively Parallel Processing (MPP) systems. SMCs are multiprocessors typified by their support for memory models that aren't *strongly ordered*. The weakened memory model enhances the scalability of SMCs. The processing elements (PEs) can be off-the-shelf uniprocessor chips with private caches that buffer load/store accesses to both private memory and a common pool of shared memory. This pool can be centralized as in the IBM Shared Memory System POWER/4™ [IBM93] or distributed as in the Convex SPP.

1.3 IBM Shared Memory System POWER/4

The IBM Shared Memory System POWER/4 combines four IBM POWER RISC processors into a single system. Each processor has private access to one local memory card (from 16 to 128 MB) and an IBM Micro Channel I/O Bus. In addition, each processor has shared access via a non-blocking switch, to up to 896 MB of shared memory (without cache coherence controls) and an Atomic Complex that provides semaphore operations for serialization and interprocessor communications. Figure 1 shows the system overview.

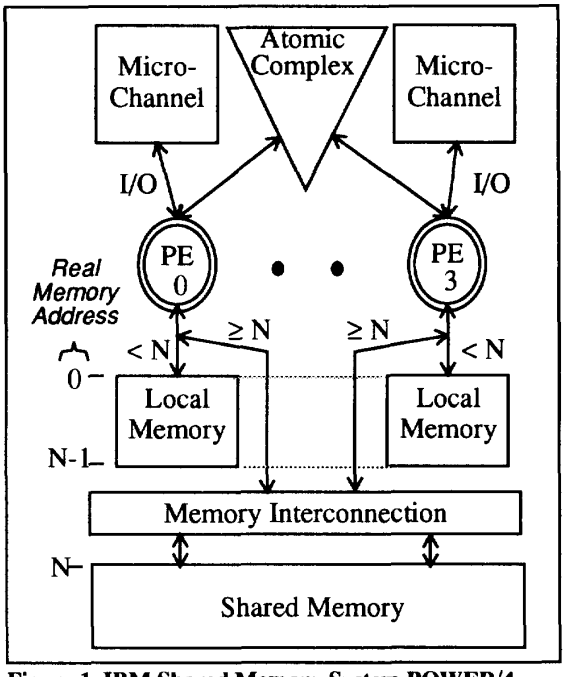

Figure 1. IBM Shared Memory System POWER/4

Each PE in the POWER/4 has an 8KB instruction cache and a 32KB data cache. The data cache is four-way set associative with 512 cache blocks of 64 bytes each. The cache management operations are those provided by the unmodified PEs. These include instructions to:

- *Invalidate* an address. The contents of the cache block containing the address are discarded.
- *Store* an address. The cache block is copied to real memory.
- *Flush* an address. The cache block is invalidated after it has been stored if it was dirty.
- *Synchronize* cache. This instruction delays until all of the in-flight store and flush operations have completed.

2. User Data Coherence

While it may be possible to rewrite all applications to run on an unordered model, our goal was to be able to run existing applications which were written assuming a strongly ordered model of memory.

2.1 Coherence Conditions

Coherence of memory accesses requires that all accesses to shared memory always retrieve the result of the latest write to that memory. When caches are being used, the following *Coherence Conditions* will provide memory coherence:

1. Multiple processors can have read access to the same memory block provided no processor has write access.
2. At any one time only one processor can have write access to a specific memory block.
3. A modified memory block is stored to real memory before another processor receives an access right for that memory block.
4. A processor's first access to a memory block that has been modified by a different processor is satisfied from real memory, not from its cache.
5. If a processor is executing with write access to a memory block, then it can also have read access; but no other processor can have read or write access to that memory block.

2.2 Page Coherence

The Coherence Conditions require that accesses to shared memory on one processor change the way in which memory is cached and accessed by other processors. For example, if processor A is reading a shared memory block and processor B attempts to write to that memory block, we must prevent processor A from reading again until the write from processor B is stored back from processor B's cache into shared memory. In addition, processor A's cached version must be invalidated to force the next reference to get a new copy of the modified data from memory.

We can use the page table to control the access to memory that is shared between user programs. On the POWER/4, 64 memory blocks (64 bytes each) are wholly contained in each page (4KB). Each processor maintains its access level to each page as either None, Read Only (R/O), or Read/Write (R/W). Figure 2 shows the per-processor state diagram for each shared user page.

Shared.Read and *Shared.Write* are synchronous requests that are sent to the other processors that may have the shared page mapped. Processors that receive a *Shared.Read* notification for a page they have mapped as R/W must store the page's memory blocks and reduce all

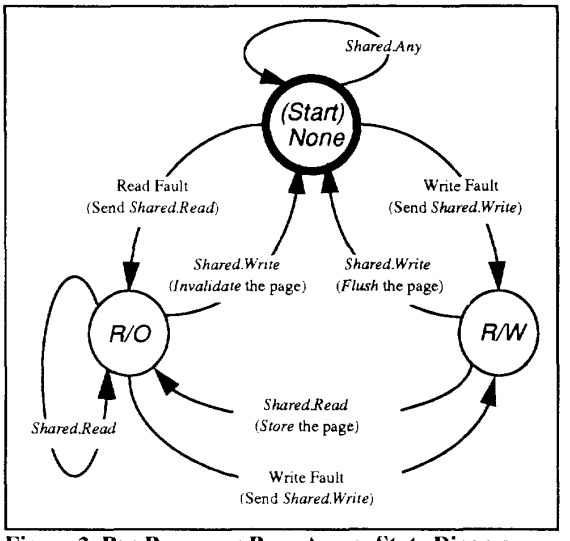

Figure 2. Per-Processor Page Access State Diagram for Coherence of Shared User Pages

access on their processor to R/O before replying. A processor that receives a *Shared.Write* must flush the page if it is mapped R/W, then reduce all access to None before replying.

Thus in our example, both processor A and processor B initially have no access to the shared data. When processor A attempts to read the data, a page fault occurs. The operating system sends a *Shared.Read* request to other processors, changes its page table to allow read access, and resumes execution of the user program on processor A. Later when processor B tries to write into the shared data, a page fault occurs on processor B. The operating system sends a *Shared.Write* request to other processors. Processor A invalidates its cache copy of the shared data, and changes its page table to deny further access to the shared data. Processor B waits for the reply from processor A and then changes its page table to allow write access to the shared data. If processor A reads the data again, it will again fault, sending a *Shared.Read* request to processor B which will store the shared data from its cache into memory, reducing the allowed access in its page table to read access. Processor A waits for the reply from processor B and then changes its page table to also allow read access to the shared data.

2.3 Performance

Providing cache coherence via the page protection mechanism has two drawbacks compared to hardware solutions (which are memory block-based). First, the *latency* for software page fault handling is higher than the latency for hardware coherence schemes because of the context switching required for handling the page/protection fault. In addition, if page tables are maintained in local memory that can only be accessed from the owning processor, messages must be used, further increasing the overhead.

Second, *false sharing* increases since the shared unit size increases [EK89] from a memory block (64 bytes) to a page (4K). As an example, consider a pair of cooperating processes, executing on different processors, sharing data that resides in two different memory blocks that are in the same page. Assume that one process is updating data in one memory block while the other is simultaneously updating data in the other memory block. In a hardware coherent environment no coherence logic is invoked since the operations are in different memory blocks. But in the page-based environment, each access will likely cause a page fault since the updates are taking place to the same page. Ownership (or the right to write to the page) could switch back and forth between the two processors on every access - sometimes called the "Ping-Pong" effect.

One technique to reduce the "Ping-Pong" effect is to provide a processing window during which time access to the page won't be taken away. Any other processes faulting during this window would be delayed.

Application awareness of the memory block size (in this environment the size of a page) can be used to reduce the "Ping-Pong" effect [LF92] and the coherence overhead [AH91][JD92]. Applications which manipulate rows or columns of a matrix in parallel on different processors can allocate the units of work in multiples of the page size to reduce the cost of software coherence.

3. OS Coherence

3.1 Mach 3.0 Microkernel

For this project, we decided to modify an existing operating system to run on a prototype for the IBM Shared Memory System POWER/4. The operating system was Mach 3.0 from Carnegie-Mellon University. The Mach 3.0 system [Acc+86] was selected because of the manageable size of its multiprocessor-enabled microkernel and the pre-existence of a port to a (uniprocessor) IBM RISC System/6000.

At the heart of a Mach-based operating system is the Mach microkernel which executes on the bare hardware and exports a machine independent interface to its users. What are normally considered the typical operating system services are layered above the Mach microkernel as a set of servers that run in user mode. Since servers provide most of the traditional system services, porting an "operating system" that runs on Mach to a different

platform is simplified because the Mach microkernel itself hides most of the machine dependencies.

Mach provides BSD functionality via a user level server -- a single Mach task with multiple threads. Since user tasks receive a coherent view of memory (Section 2), the BSD server can run unmodified on a SMC -- only the microkernel must be adapted to execute with non-coherent memory, reducing the amount of work to be done. Figure 3 shows the relationship between the microkernel and its user tasks.

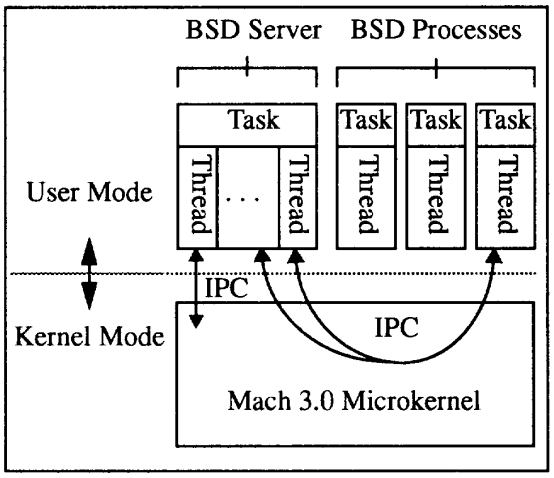

Figure 3. Mach 3.0 Structure

3.2 Thread-Based Model

The microkernel was adapted to non-coherent memory by adhering to the Coherence Conditions listed above with one major exception. The microkernel is structured as a set of kernel threads which may execute on different processors. For the microkernel, each thread is considered a separate.processor for the Coherence Conditions.

3.3 Mach Access to Shared Data

Processes that cooperate by sharing data, even on coherent machines, must do so in a way that guarantees data integrity. Mach has been designed to run on multiple processors and uses locks to provide mutual exclusion for critical sections [Bla+91]. Mach uses different locks for different data items. This allows a high degree of parallelism since the same critical section can be executing simultaneously on different processors against different instantiations of a data structure.

3.4 Preserving Data Integrity

The Coherence Conditions require controlling access to individual memory blocks. Since Mach uses locks for mutual exclusion, the data items protected by a specific lock cannot reside in the same memory block with data protected by a different lock. Specifically, shared data items are partitioned by their access (in Mach's case, locking) protocol; data items protected by exactly the same protocol form a partition. Items in the same partition can be packed into the same memory block(s) and any one memory block cannot contain data from multiple partitions. Although there are optimizations to allow some of these partitions to be combined[1], this solution is reasonably efficient and is easy to compute.

Many data items are partitioned naturally by categories that more broadly describe how they are accessed. These categories also define the options available for placement in private or shared memory and the most restrictive protection that can be assigned. Table 1 summarizes these issues.

An example of the process used to partition the shared data by access protocol (the R/W Multiple Threads category) comes from the zone structure used in the kernel. The zone structure manages a pool of quickly allocatable kernel memory. One zone structure is allocated for each pool. Figure 4 shows the source for the zone structure (as modified for memory block partitioning).

Access to all the members in part A is protected by locking either zone_S_lock (a Mach simple lock) or zone_C_lock (a complex lock) based on whether the memory for this zone is pageable. So the members in part A, all having the same access protocol, form one partition. In Mach, a complex lock contains a simple lock to protect its data structures, so zone_C_lock (section B) is in a partition by itself. Finally, next_zone (section C) is a link field that joins together all of the instantiated zone structures. The queue header and the next_zone fields from all the zone structures are protected by a separate lock and form a third partition.

Each instantiation of a zone requires the allocation of three memory blocks. The MBLOCK_PAD macros have been inserted in the structure to allocate enough space to force the next element to a memory block boundary. This assumes that the structure is block aligned. To enforce this alignment, the internal Mach memory allocation routines were changed to always allocate memory on memory block boundaries.

Partitions of statically allocated external variables must also be memory block aligned and separated. To accomplish this requires an extension to the C

1. Consider a system with two locks, L_A and L_B, where some data items are protected only by L_A, some by L_B, and the remainder by holding L_A and L_B simultaneously. The data is partitioned into three sets labeled P_A, P_B, and P_{AB}, respectively. Let P_{AB1}, P_{AB2}, P_{AB3} be any partitioning of P_{AB}. Then an optimization for this system is $P_A \cup P_{AB1}$, P_{AB2}, $P_B \cup P_{AB3}$.

```
typedef struct zone {

    struct slock zone_S_lock;  /* generic lock */
    int          count;        /* Number of elements used now*/
    vm_offset_t  free_elements;
    vm_size_t cur_size;        /* current memory utilization */
    vm_size_t max_size;        /* how large can this zone grow */
    vm_size_t elem_size;       /* size of an element */
A   vm_size_t alloc_size;      /* size used for more memory */
    boolean_t  doing_alloc;    /* is zone expanding now? */
    char       *zone_name;     /* a name for the zone */
    unsigned int
        pageable :1,           /* zone pageable? */
        sleepable :1,          /* sleep if empty? */
        exhaustible :1,        /* merely return if empty? */
        collectable :1,        /* garbage collect empty pgs */
        expandable :1;         /* expand zone? */

    MBLOCK_PAD(sizeof(struct slock) + ... + sizeof(int))

B   lock_data_t   zone_C_lock; /* Lock for pageable zones */

    MBLOCK_PAD(sizeof(lock_data_t))

C   struct zone    *next_zone;    /* Link for all-zones list */

} *zone_t;
```

Figure 4. The Mach *zone* structure

programming language that allows alignment requirements to be specified and passed to the loader (as in GNU CC). Since this was not available in the compiler being used for the port, all external declarations of shared data were changed to definitions and moved to assembly code files where alignment facilities could be used.

3.5 Preventing Data Staleness

Flushing Strategy: The approach used in this project allows data to remain in the cache between critical section executions. Each time a critical section is entered, shared data is flushed before it is touched, forcing the next reference to come from memory. Modified data is stored to memory prior to leaving the critical section thus ensuring that a valid copy exists should it be needed on another processor. This approach was used because it could be (but hasn't been) extended to have staleness "awareness", much like the *Version Verification Scheme* proposed by Tartalja and Milutinovic [TM91], which could be used to avoid the flush when cached copies aren't stale.

Static Data: Some of the shared data accessed within a critical section may be static in nature. When the lock protecting the data is instantiated, the addresses and lengths of these regions are well known and don't change.

Data Category	Memory Placement	Virtual Storage Mapping	Partitioning
Read Only Constants	Local	R/O	Unpartitioned R/O data can be distributed across all other partitions.
Read Only After Initialization	Shared	R/O	
Write one processor Read same processor	Local	R/W owner, None for others	One per processor
Write one processor, Read many	Shared	R/W owner, R/O for others	
R/W Single Thread	Local if thread cannot migrate. Shared otherwise	R/W on active processor, None on others	One per thread
R/W Multiple Threads	Shared	R/W	By access protocol

Table 1. Management of Kernel Data

The Mach functions used for initializing locks were changed to include the specification of up to two data areas which are recorded in the lock structure itself. The lock and unlock functions were changed so that these areas were flushed and stored appropriately.

Dynamic Data: Some of the data associated with locks is dynamic in nature, changing as the state of the system changes. A common example is a lock protecting a queue header (static) and all the queue chain fields of those structures linked on the queue (dynamic). There are two approaches for managing the coherence of dynamic data. The first, which we call the *encapsulated* approach, is to ensure all the data is coherent as part of the lock manipulation, much like the lock's static data. This requires an awareness of the structuring of the shared data, which could be provided by a routine(s) whose address is passed when the lock is initialized. Its address would be maintained in the lock's data structure, and it would be invoked as the lock is manipulated. This would be similar to a virtual method in an object oriented language.

The advantage of the encapsulated approach is that it is tied to the data structures rather than the program logic.

When adapting existing systems for cache coherence, intrusion into the programming logic is minimized and logic changes don't affect the methods that maintain coherence. The disadvantage of this approach is that all of a lock's shared data is made coherent every time the lock is used, even though in many cases, only a small portion of the data is manipulated in any one critical section.

Another approach, which we call the *distributed* approach (the one used in this project), is to insert the cache management calls in-line with the code that touches the shared data. Each critical section will have statements added to invalidate shared data before it is referenced and to store data that has been modified.

The advantage of the distributed approach is that only those pieces of data that are actually touched while the lock is held are made coherent. A disadvantage of the distributed approach is that it is prone to over-flushing the data that is modified. The most common cause of this over-flushing is that subroutines may not be aware that some of the data that they touch may already be coherent, requiring it to also ensure its coherence.

3.6 Other Issues

The goal of this project was to determine the feasibility of adapting an operating system to a non-coherent environment and to roughly measure performance. This allowed for design trade-offs that would be unacceptable in a production environment but that had little impact on our results.

Task/Thread Binding: Mach extends the Unix process model by separating it into a task with possibly multiple threads. The thread is the unit of scheduling. Each thread has its own state that contains such things as its instruction counter, registers, and stack. The task is the basic unit of resource allocation, and collects the common state for its threads. A task includes a paged virtual address space and protected access to system resources, including its memory mapping (all of a task's threads share the same virtual storage mapping), processors, and port capabilities. The traditional notion of a Unix process is represented by a task with a single thread.

The changes to the machine dependent code for multiprocessor execution were minimized by a design decision that restricts all threads from the same user task to execute on the same processor. In a system supporting only BSD "processes", this design decision only impacts the BSD single server, which is the only user task with multiple threads.

Thread Migration: In Mach, user threads are free to migrate between processors. We changed Mach changed so that whenever a new task is created, a processor is selected (via round-robin) to which all threads created under the task are bound. This means that there is no load balancing built into this implementation. Threads are assigned permanently to a specific processor (threads are, however, dispatched on the "master processor" for some portion of I/O processing). The mechanism to reassign a task and its threads to a different processor was implemented, but no policy was developed to cause the reassignment to occur.

These restrictions don't hold for kernel threads, some of which are bound to the "master" processor while others are free to migrate.

Thread migration is a problem in a non-cache coherent machine. The kernel's shared data that is protected by simple locks will remain coherent (threads can't yield or block while holding a simple lock -- so they can't be migrated while accessing this data). Kernel threads that can migrate can lose coherence to other types of data (e.g. data protected by complex locks, data on their stack).

For this reason, the kernel has been modified to keep track of the last processor on which each thread executes. Each time a thread is dispatched, this is examined, and if the thread has migrated, a message is sent to the previous processor. Both the previous and the current processors invoke a routine to flush the entire contents of their data caches. This guarantees coherence.

I/O Management: Each processor in the POWER/4 system has non-shared access to its own I/O bus. It would have been feasible, but difficult, to merge all the processors' devices into a global device name space. Instead, only the devices from one processor, called the "master", are made visible. All I/O operations are funneled through the "master".

This implementation allowed the device drivers and interrupt handlers to execute, for the most part, without change. Stubs were added to the device drivers to bind the calling thread to the master until the I/O is queued, at which time the thread is returned to its previous state, bound again to its previous processor if appropriate.

4. Cache Violation Detection

4.1 Traditional Tools

When the system fails in such a way that a coherence problem is suspected, locating the problem is usually a matter of reviewing all the code that deals with the (possibly) incoherent data structure, looking for failures to flush or store. Unfortunately, it's often difficult to determine the cause. By the time the problem manifests itself in a failure, the source of incoherence may be very hard to identify.

An on-line debugger may be of little value for determining the type and source of the failure. The execution of the debugger can interfere with the cache state, hiding the fact that a stale cache line caused a

problem. Several times in debugging a problem the only approach to isolating the failure was to modify the source code to report the execution time values of data and to rebuild the kernel.

During the first three months of testing only three coherence problems were resolved. It became clear that it might take years of work before the modified system would run. When the fourth problem appeared, a decision was made to either develop some tools and techniques to enhance debugging, or to abandon the project. Fortunately, a mechanism that could be used to detect coherence "violations" was developed.

4.2 Violation Detection

The foundation for our mechanism for detecting cache violations is the "promotion", through macros, of the memory block size to be the same as the page size. Memory allocations occur on page boundaries. Partitions that were private to a memory block now occupy an entire page. Since most partitions fit into a single memory block, this results in internal fragmentation of 98% (63/64). Low-level operating system code was written which maintains state information for every page (partition) of shared kernel data. This code is called by the macros that are used to invoke the cache management operations and manipulates page table access and state information for the shared data being flushed. Page table access to these pages is controlled, as in the page coherence model, on either a thread or a processor basis so that accesses to this data cause program traps. These traps are handled by low-level code which checks and maintains the state of the shared data, reporting inconsistencies as cache (protocol) violations.

Thread-Based Detection: The first tool developed maintains state information on a per-thread basis. Each thread's initial access to shared data pages is set to None. The macros used to invalidate and store sections of data were modified so that state information could be maintained about their use. At page fault time, the state is examined to ensure that the thread flushed the data, then its access is set to either R/O or R/W based on the type of fault. When a block is stored, the thread's access is changed to R/O.

State transitions for one thread's access to a page cause the system to examine, and possibly modify, other threads' states, reporting violations as appropriate. Consider an example where $thread_i$ invalidates some data and then blocks waiting for some event. If in the interim another thread invalidates and modifies this data, the state information for $thread_i$ is reset. If $thread_i$ is resumed and touches the data (without another invalidate), a violation will be reported. In all, there are 41 types of protocol violations that can be detected in this scheme. Those interested in seeing the page state transition diagram for this facility are referred to [Roc93].

Probably the most significant benefit of this scheme is its ability to detect most, but not all, protocol violations when running on a uniprocessor. The strength of this mechanism is that those violations it can detect are reliably reported whenever they occur.

The most significant limitation of the thread-based mechanism is that it has no awareness of locks and the data they protect. It does not always detect protocol failures that involve the improper sequencing of locking and coherence operations. This is a significant class of problems that was revealed only by implementing the Processor-Based detection as described in the next Section.

Processor-Based Detection: After further testing using the thread-based detection scheme revealed no more problems, the system was moved to the prototype machine. Unfortunately, the system still suffered what appeared to be data coherence problems. Recognizing that some classes of problems may have gone undetected by thread-based detection, a processor-based detection scheme was developed.

In this mechanism each processor maintains information on a per-page basis. Whenever a processor's access level to a page increases (None $\rightarrow$ R/O $\rightarrow$ R/W) it sends a Cross Inquiry IPC message to the other processors. The receiving processors check to see if this page has been modified since it was last stored, and if so, a violation is reported (by both the sender and the receiver). This is one example of the 10 different protocol violations that can be detected using this scheme.

4.3 Violation Detection Results

The effort expended to develop the testing environment was well spent. Thirty coherence problems were detected and resolved in the three weeks of testing that were required before the test cases ran cleanly. Some were of such an obscure nature that it is doubtful that they would have been found otherwise.

This environment required an average of 0.5 days to identify and resolve one problem compared to an average of one month using "traditional" debugging methods. For the 30 problems detected, this represents a savings of 29.5 months.

Memory Requirements: A result of the internal fragmentation is that kernel data requires 64 times as much real memory. This required changing the kernel memory pools and increasing the real memory of the test machine to 64MB. The size of the kernel image grew from 480KB to 3.6MB.

Performance: The performance of the system with violation detection enabled is abysmal; most operations are approximately 1,000 times slower. This results from the

page faulting activity. Each time a thread is dispatched, all the kernel data is mapped to None. The first reference to a page causes a page fault. If the first reference to a page is a read, then the page is mapped R/O, which causes the first write access to cause a fault as well. When an invalidate is issued, the page access is set to None, and the thread could begin faulting again. Every time a thread suspends or blocks, the status of the pages it touched are examined to determine whether or not they have been stored back to real memory. In addition, the access to each kernel page is set to None to prepare the state for the next thread.

With the violation detection facility enabled, it took over three days to boot the system to a login prompt. For this reason it was necessary to add a facility to turn on the violation detection at will, rather than automatically at boot time. Each time the system was booted, the detection code was turned on only after it progressed to the point reached in the previous debugging session.

5. Effort

The most effort of this project (five person-months) was spent on the Mach microkernel (machine independent) code, categorizing the data and adding the appropriate flushing logic. The original 95,000 source lines in release MK67 required 4,000 new and modified lines of code. Of these, 2,754 were cache flushing macros and 140 were padding macros used for aligning data in 30 different data structures. There were 82 different locks defined which represents a measure of the partitioning requirements. No count of the total number of external variable definitions was made, but 483 of them were moved to assembly source files to accomplish correct virtual storage mapping and correct memory block alignment and grouping.

Approximately four person-months of effort went into changes to the machine dependent code. The original port did not support multiprocessing systems, and most of the effort was spent here. The original base of 30,821 lines of code grew to 40,023; an increase of 9,202 lines. Of these, 297 were cache flushing macros. There were 140 variable definitions that were moved to assembler files.

Developing the cache violation detection facility required approximately one month of effort. Of this, three weeks were needed to develop the thread-based model; one week for adding the processor-based code. The effort resulted in approximately 1,300 lines of C code.

The testing phase lasted approximately four months. Most of this was time spent on the POWER/4 Prototype prior to the implementation of the violation detection code (three months). After the detection code was developed, three weeks were spent testing on a uniprocessor, and one week was spent on the multiprocessor.

There is little chance that this project would have been completed without the development of the detection tools. Cache coherence bugs do not lend themselves to resolution through conventional debugging techniques.

6. Performance

6.1 Benchmarks

Two benchmarks were used to measure the performance of the operating system. Since the primary objective of the project was to demonstrate correct function and not performance, a random selection from the SPEC benchmark suite was made.

The Espresso benchmark executes espresso 2.3, an integer benchmark from U.C. Berkeley, using the input file bca.in. Li is a CPU intensive benchmark implementing a Lisp interpreter, based on XLISP 1.6 and written in C. Version 1.0 of Li, developed at Sun Microsystems, was used.

Since runs with each combination of the four processors were desired, and since no load balancing policy was implemented in the system, each benchmark consisted of a shell script that executes, in the background, twelve instances of the same program.

6.2 Base vs. Multiprocessor Enabled Code

The data structures and operations needed to run on a multiprocessor (the locks, flushing and so on) are written as macros in Mach. By changing preprocessor definitions, it is possible to "compile out" all the code that is necessary only when executing on a MP, producing a uniprocessor version of the system. An interesting measurement compares this uniprocessor version of the operating system (labeled UP) with the version that has been enabled for cache coherence and multiprocessor support (labeled MP) running on a the same uniprocessor. The UP version can take advantage of the fact that it is running on a uniprocessor. It does not need to perform any locking, flushing, and IPC queueing operations. Table 2 shows the results of executing both benchmarks on an IBM RISC System/6000 model 530 with 128MB of real memory.

The *real* fields represent the elapsed (wall clock) time for the benchmark. The *user* and *sys* fields show the accumulated amount of CPU time directly spent executing the job as divided between user mode and kernel (system) mode. The CPU Share field is calculated as *real* / (*user* + *sys*) and represents the portion of the CPU cycles available for the duration of the benchmark that were used directly in the execution of the jobs.

This comparison shows that the MP version provides overall throughput of between 2/3 and 3/4 that of the unmodified (UP) version. The most significant change

RISC System/6000 Model 530	Espresso (bca.in)		Li (Lisp Interpreter)	
	UP	MP	UP	MP
real (secs.)	228.5	338.5	4,301.6	5,550.6
user (secs.)	220.3	243.4	4,229.4	4,698.9
sys (secs.)	4.9	61.3	67.0	765.7
real (hh:mm:ss)	3:49	5:39	1:11:41	1:32:31
CPU Share	99%	90%	100%	98%
Speed Up	1.00	0.68	1.00	0.77

Table 2. Unmodified (UP) vs. Modified (MP) On a Uniprocessor

between the two systems is the *sys* component. It represents the time this task was executing the kernel code which, of course, was modified for cache coherence and multiprocessor support. This is a factor of twelve larger for the MP system. Although no instrumentation was included which would help quantify the causes of the increase, there are several contributing factors:

- **Locking operations.** The UP version has locks and locking operations "compiled out" of the kernel. The MP version does not. Each lock/unlock operation requires acquiring a (simulated) semaphore, flushing the cache line containing the lock, testing the lock's value, setting the lock value, storing the cache line, and releasing the semaphore.
- **Cache misses.** When the kernel first accesses shared kernel data after acquiring the appropriate lock, it flushes it to eliminate the possibility of stale data. This unfortunately guarantees that the next access will be a cache miss.
- **Cache store with synchronize.** Before the kernel releases a lock it must store any changes to real memory and synchronize the operation to be sure the lock is released only after the data is stored. This is an even longer operation than required as a result of a cache miss.
- **MP code.** The Inter-Processor Communication (IPC) code is not disabled when only running on a uniprocessor. Any requests that need broadcasting are still added to the IPC queue. Of course, the item is immediately dequeued since the count field will be zero, but the overhead of building and queueing is still there.

6.3 Scalability

The processor-based model implemented for user tasks does not lend itself to low overhead, concurrent accessing of shared data for cooperating processes. The cache cross interrogate (XI) protocols require less overhead when implemented in hardware rather than software (no context switching). The unit of protection is a page (rather than a cache block for hardware-based coherence schemes) which can lead to more XI traffic because of a higher incidence of false sharing.

The system, therefore, was measured for its scalability in a throughput environment using the two benchmarks described earlier. It is of value to note that even though the programs that comprise the benchmark are not themselves cooperating user processes, the BSD single server does share memory with user tasks. Each task has a three page data area that it shares with the BSD server. The server task and the emulator code in the user task share this area in R/W mode in an effort to reduce message passing through the kernel.

Table 3 shows the results of executing these benchmarks on the POWER/4 Prototype machine configured as a 1, 2, 3, and 4-way multiprocessor. Each processor had 64MB of local memory with access to 128MB of shared memory.

The most significant results from these benchmarks is that the system appears to scale well as a batch throughput machine.

As mentioned earlier, there is no load balancing policy implemented for the system; tasks are assigned permanently to a specific processor for the duration of their execution. This means that in each of the configurations, one of the processors is not only executing the same number of benchmark jobs as the others, but in addition, it is executing all of the BSD server code. Since the benchmark is not considered complete until all twelve jobs are finished, there is some amount of time near the end of the benchmark where all the processors, except the one that's running the BSD server, are idle. The availability of this excess capacity has not been included in the throughput analysis.

7. Conclusions

The most significant result of this project is that it demonstrates that an existing multiprocessor operating system can be modified to function correctly on a non-cache coherent multiprocessor that caches shared data while exporting a coherent, strongly ordered, symmetric multiprocessor view to users. Not only can it be accomplished, but it required only 14 person-months of effort to complete.

The second most significant result comes from the potential scalability of the resulting system. On a four processor SMC the system provides between 3.26 and 3.94 times the throughput of a single processor. That the system performs so well is especially promising since there was no priority given to efficiency issues during the development. Once the system was running, no effort was

POWER/4 Prototype	Espresso				Li			
	1 CPU	2 CPUs	3 CPUs	4 CPUs	1 CPU	2 CPUs	3 CPUs	4 CPUs
real (secs.)	1,005.3	528.5	382.1	308.5	18,264.3	9,015.5	6,127.3	4,635.6
user (secs.)	660.3	646.5	646.7	653.1	12,504.2	12,229.2	12,261.6	12,360.0
sys (secs.)	199.3	241.6	251.3	259.2	4,029.0	4,711.5	5,267.8	5,055.2
real (hh:mm:ss)	16:45	8:49	6:22	5:09	5:04:24	2:30:16	1:42:07	1:17:16
CPU Share	86%	168%	235%	296%	91%	188%	286%	375%
Speed Up	1.00	1.90	2.63	3.26	1.00	2.03	2.98	3.94

Table 3. Benchmark - Multiprocessor Results

made to tune it or to bias the results by selective reporting. The benchmarks themselves were randomly selected.

A major conclusion of this effort is that although it is possible to construct a system that tolerates non-cache coherent hardware, it is nearly impossible to test it without access violation detection assistance. In our case, we developed software techniques that were sufficient, but the elongated execution times were so severe that it would be difficult to do thorough testing without special hardware assistance.

We have shown that an N-way SMC can be made to appear to the user as an N-way symmetric multiprocessor. More generally, for scalability or redundancy purposes, these N processors can be subset into M partitions with each partition executing as a symmetric multiprocessor, forming an M-node cluster.

8. References

[Acc+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development," Proceedings of the Summer 1986 USENIX Conference (1986): 93-112.

[AH91] S.G. Abraham and D.E. Hudak, "Compile-Time Partitioning of Iterative Parallel Loops to Reduce Cache Coherency Traffic," IEEE Transactions on Parallel and Distributed Systems vol. 2 (1991): 318-328.

[Bla+91] D.L. Black, A. Tevanian, Jr., D.B. Golub, and M.W. Young, "Locking and Reference Counting in the Mach Kernel," 1991 International Conference on Parallel Processing VII, Software (1991): II-167 - 173.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering In Multiprocessors," Proceedings of the 13th International Symposium on Computer Architecture (1986): 434-442.

[EK89] S.J. Eggers and R.H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II, 1989): 257-270.

[Gha+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," Proceedings of the 17th Annual International Symposium on Computer Architecture (1990): 15-26

[IBM93] IBM Shared Memory System POWER/4 User's Guide and Technical Reference (IBM Corporation, 1993).

[JD92] Y. J. Ju and H. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," Languages and Compilers for Parallel Computing. Fourth International Workshop (1992): 344-358.

[Lam79] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers (Sept. 1979): 690-691.

[Len+90] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," Proceedings of the 17th International Symposium on Computer Architecture (1990): 148-159.

[LF92] M. Lu and J.Z. Fang, "A Solution of the Cache Ping-Pong Problem in Multiprocessor Systems," Journal of Parallel and Distributed Computing vol. 16 (1992): 158-171.

[Roc93] R.L. Rockhold, "Software-Based Cache Coherent Operating Systems." Ph.D. dissertation, Florida Institute of Technology, 1993.

[TM91] I. Tartalja and V. Milutinovic, "An Approach to Dynamic Software Cache Consistency Maintenance Based on Conditional Invalidation," Proceeding of the 25th Hawaii International Conference on System Sciences vol. 1 (1991): 457-466.