The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM

Alan L. Cox
Robert J. Fowler
Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract

PLATINUM is an operating system kernel with a novel memory management system for *Non-Uniform Memory Access* (NUMA) multiprocessor architectures. This memory management system implements a *coherent memory* abstraction. Coherent memory is uniformly accessible from all processors in the system. When used by applications coded with appropriate programming styles it appears to be nearly as fast as local physical memory and it reduces memory contention. Coherent memory makes programming NUMA multiprocessors easier for the user while attaining a level of performance comparable with hand-tuned programs.

This paper describes the design and implementation of the PLATINUM memory management system, emphasizing the coherent memory. We measure the cost of basic operations implementing the coherent memory. We also measure the performance of a set of application programs running on PLATINUM. Finally, we comment on the interaction between architecture and the coherent memory system.

PLATINUM currently runs on the BBN Butterfly Plus^{TM} Multiprocessor.

1 The Need for Transparent Management of Non-Uniform Memory

PLATINUM is an operating system kernel designed to be a platform for research on memory management systems for *Non-Uniform Memory Access* (NUMA) multiprocessor architectures, those in which the distributed, shared memory of the machine can be referenced by any processor on the machine, but the cost of accessing a particular physical location varies with the distance between the processor and the memory module. The name "PLATINUM" is an acronym for "Platform for Investigating Non-Uniform Memory". Its purpose is the experimental evaluation of a software implementation of a *coherent memory* abstraction on top of non-uniform access physical memory architectures. PLATINUM runs on BBN Butterfly Plus^{TM} Parallel Processors.

One can achieve impressive speedup due to parallelism on a NUMA multiprocessor, but unfortunately this can entail a considerable effort. Because remote memory references are an order of magnitude more expensive than local references and because remote references are subject to several forms of potential contention, the physical location of data is critical to performance. On the BBN Butterfly^{TM} Parallel Processor, a popular and productive way to deal with the problem of shared data location is to avoid the question by using libraries [20] and languages [26] that support message passing. When using a non-uniform access memory directly, however, one has to deal with data locality. This programming of data locality is reminiscent of the explicit management of memory hierarchies using overlays: attaining performance can be non-intuitive and can depend upon dynamic properties of program execution; worse, it has to be done explicitly by every application programmer. The importance of this tuning is such that a programmer can expend far more effort on "programming the memory architecture" than in solving the original application problem.

Our goal is to explore the possibility of achieving performance comparable to that of hand-tuned programs with a simple, easy-to-program shared-memory model. It is our hypothesis that it is crucial to present users with a simple model of shared memory implemented so as to attain good parallel performance on applications written in a natural programming style. The coherent memory model implemented by PLATINUM is an exercise in doing this transparently in an operating system kernel on top of an existing NUMA multiprocessor. Because we wish to explore the limits of this approach, PLATINUM assumes neither special architectural support nor extensive language-specific assistance from a compiler. We do believe that these are vital in the long run and exploring these issues is a part of our long-term research interests.

NUMA multiprocessor organization leads to memory management design choices that differ markedly from those that are common in systems designed for uniprocessors or UMA multiprocessors. If two or more processes on a uniprocessor are sharing read-only data such as a common code segment, it is wasteful to allocate multiple private copies. Such replication is expensive in terms of the number of physical pages used and in terms of the expense of copying the data. For example, to reduce this expense, the implementation of Mach [23] minimizes the amount of data-copying and replication through the use of copy-on-write and other

This work is supported in part by U. S. Army Engineering Topographic Laboratories research contract no. DACA 76-85-C-0001, in part by ONR research contract no. N00014-84-K-0655, and in part by NSF research grant no. CCR-8704492.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 089791-338-3/89/0012/0032 $1.50

techniques.

In contrast, extra data motion in the form of replication and migration can yield greatly improved performance on a NUMA machine. Placing data in the local memory of a processor that is using it decreases memory access latency. More importantly, a processor accessing local data is not performing remote operations that contend for remote memory modules and for the processor-memory switch. These two factors also motivate the use of caches in bus-based multiprocessors [13]. The advantages of replication and data motion distinguish the problem of managing memory on a NUMA machine from the same problem on uniprocessors and *Uniform Memory Access* (UMA) multiprocessors.

PLATINUM's implementation of coherent memory replicates and migrates data to the processors using it, thus creating the appearance that memory is uniformly and rapidly accessible. The protocol for controlling this data movement is derived by extending a directory-based cache coherency algorithm using selective invalidation [7, 2]. The extension exploits the NUMA architecture by adding the option of using the remote memory access mechanism rather than replicating or migrating data to local memory on an access miss. Using remote memory access effectively disables caching on a block-by-block basis. This is crucial when write-shared data is modified at fine temporal and spatial granularities because the overhead of executing a coherency protocol can be more expensive than not caching. With the large block sizes and overheads associated with software-assisted caching, the effect can be especially bad. This is a critical distinction between NUMA memory management in PLATINUM and the software caching of Li's Distributed Virtual Memory [22] or the software-controlled caching of the VMP Multiprocessor [9, 8].

Because the measured performance of real applications is a far better indicator of the success of a system than analytic predictions, simulations, or simplified experiments, PLATINUM provides enough of a general-purpose application environment to support such programs. We are actively building a library of applications designed to test the performance of PLATINUM with a variety of programming styles that use different memory access patterns. The results are encouraging. Figure 1 plots the speedup of a program that simulates Gaussian elimination without pivoting on dense matrices. In this case the input is 800 by 800. This particular problem was chosen because it was used in performance studies of programming systems [10, 18] on earlier versions of the Butterfly. It simulates Gaussian elimination in the sense that it uses integer rather than floating-point operations, thus emphasizing the relative impact of memory performance with respect to the speed of arithmetic operations.

The design of PLATINUM targets factors such as ease of programming and performance, since these are the primary criteria by which the coherent memory abstractions should be judged. While other issues such as security, protection, and long-term storage have been considered in the abstract design, they have received only cursory attention in the current version.

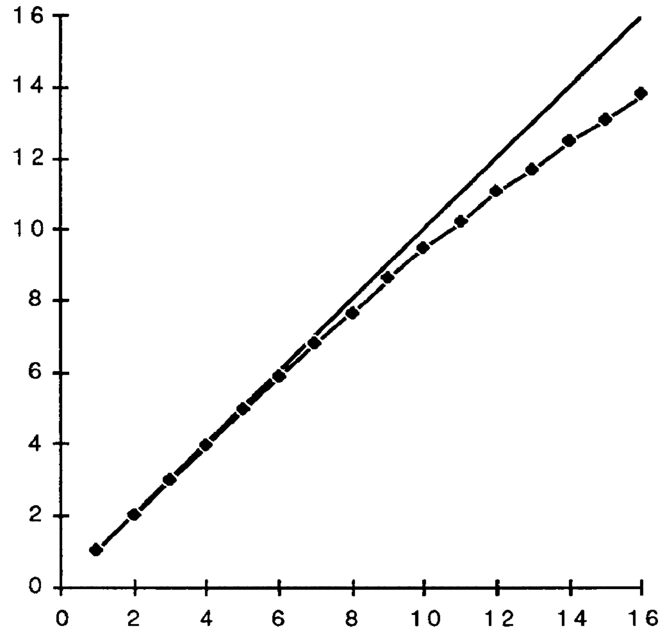

Figure 1: Gaussian Elimination (Speedup vs. Processors)

1.1 PLATINUM Programming Model

Since our goal is the exploration of transparent NUMA memory management, we use familiar abstractions and interfaces as much as possible. This decision determined the interface presented to the user and some of the internal kernel interfaces. We used the Mach [23] model of memory as the prototype because of its modularization into machine-dependent and -independent parts. Within a stripped-down version of this model, PLATINUM coherent memory is implemented as a replacement for the machine-dependent part of the memory management system.

PLATINUM exports to user programs an abstract multiprocessor model in which all primary memory accessible to user programs appears to be a fast (on average) shared physical memory module uniformly accessible from all of the processors in the system. The physical location of data in primary memory is hidden from the user. PLATINUM allocates memory in page-aligned regions. Page boundaries are not hidden, enabling the user to reduce interprocessor interference by allocating shared data with different access patterns to distinct pages.

The fundamental abstractions supported by PLATINUM are the *thread*, the *memory object*, the *port*, and the *address space*. These objects all appear in a single flat global name space.

A *memory object* is an abstraction of an ordered list of memory pages. A range of pages within a memory object may be bound to any contiguous page-aligned virtual address range of the same size. Neither the virtual address range nor the access rights need be the same in every address space. Since they have global names, memory objects are the natural unit of data- or code-sharing between address spaces.

A *thread* is a kernel-scheduled thread of control. At any time it is bound to a single processor. An explicit migration operation can move it to another location. It is, however, constrained to execute within a single address space.

An *address space* is a list of bindings of memory objects and access rights to virtual address ranges. It defines the

environment in which one or more threads may execute. The threads in a single address space may be distributed to multiple processors.

A *port* is a message queue that can have any number of senders and receivers [1]. Messages are variable-length arrays of zero or more bytes. Globally named, ports provide a communication medium usable by threads that do not share a common memory object. They also provide blocking synchronization.

Logical concurrency is realized through the use of multiple threads to implement a single application. True parallelism is realized by running those threads on multiple processors. Communication between threads can be based on either shared memory or message-passing via ports. Threads that coexist within a single address space share all of the memory objects mapped into that address space. This implies, in addition to data coherency, that these threads share a coherent view of the mappings of memory objects that constitute the shared space. A more restricted form of sharing is realized by mapping a memory object into multiple address spaces. The shared object can be accessed by all of the threads in those spaces, but the non-shared objects in each address space are protected from threads in other spaces.

A comprehensive description of the interface can be found in [11]. Given the initial successes with PLATINUM, its interfaces are being extended as required to provide added functionality and ease of programming to support larger experiments. We are also adding an instrumentation interface to the kernel to help interpret its behavior. The design is intended to make it easy to integrate PLATINUM coherent memory with Mach.

2 Organization and Implementation of the Memory Management System

A typical virtual memory system has traditionally managed a memory hierarchy consisting of a cache, a uniformly accessible primary memory, and a significantly slower secondary memory. The existence of remote primary memory on a NUMA multiprocessor adds at least one more level to this hierarchy. PLATINUM memory management is structured to separate the traditional responsibilities of virtual memory management from the additional requirements imposed by the NUMA architecture. The memory management system is constructed in three layers. The highest layer is the *Virtual Memory* system. The middle layer is the *Coherent Memory* system. The lowest layer is the *Physical Map* system.

2.1 Organization

The virtual memory system manages the mappings from virtual address ranges to memory objects and from memory objects to coherent pages (see the left side of Figure 2). The machine-independent part of Mach memory management is the prototype for this layer.

The coherent memory system is responsible for the mappings from coherent pages to physical pages. These may be one-to-many. The left side of Figure 2 shows coherent-to-physical mappings for one of the three memory objects.

The coherent memory system also guarantees the consistency of the physical pages backing a coherent page. This is implemented by extending a directory-based protocol that uses selective invalidation to maintain coherency [7]. For each coherent page the system maintains a directory of the set of physical pages backing it. A new physical page is added to the set when the system chooses to replicate the coherent page. The replication policy makes the decision between the replication of a coherent page and the creation of a mapping to an existing physical page. When a processor writes to a replicated coherent page, all but a single physical copy are invalidated and removed from the set.

The implementation of the protocol makes heavy use of the hardware memory management unit (MMU), on the Butterfly Plus a Motorola MC68851. Access rights to physical pages are potentially more restrictive than those specified by the virtual memory system in order to ensure the generation of traps by memory accesses which require action. Most transitions in the protocol are thus initiated by address translation and protection faults, and are performed by the page fault handler.

The physical map system is a simple machine-dependent page table and address translation cache management module. For each address space a physical page map (Pmap) is used to cache the compositions of the logical mappings maintained by the virtual and coherent memory systems. Each physical mapping illustrated on the right side of Figure 2 is the composition of a corresponding sequence of mappings on the left side of the figure.

2.2 Implementation Strategy

The promise of high performance, scalable parallelism using a shared-memory model of computation makes NUMA multiprocessor architectures interesting. It is therefore vital that an operating system kernel be very efficient and avoid limiting the scalability of the system. The memory management system is implemented with this in mind. Kernel operations and data structures are decentralized to provide maximum concurrency. Wherever possible, atomic memory operations are used to implement concurrent data structures. When an explicit lock is needed, the scope over which it is held is kept small to reduce the residual impact of contention between concurrent kernel operations. Remote memory accesses in critical sections are avoided, especially within the coherent page fault handler. In some cases the algorithms and data structures use several local memory accesses to avoid a single remote memory access.

The kernel address space consists of two regions, one in physical memory and the other in coherent memory. Kernel code and the data structures for the lowest kernel layers are in physical memory. These structures implement physical and coherent memory systems as well as physical device handlers. The kernel replicates its code and read-only data. Since writable data in physical memory can only have one copy, each writable page in kernel physical memory is mapped for remote access by all but its local processor.

The layers of the kernel that implement virtual memory, threads, and ports keep their data structures in the coherent memory region. Because they are in the coherent memory region, kernel stacks for threads require special handling. Otherwise, the first fault after a thread has moved would try to save the processor state on the kernel stack for the

[1] A message queue that allows multiple receivers is usually called a "mailbox". The use of "port" reveals the Mach ancestry of PLATINUM.

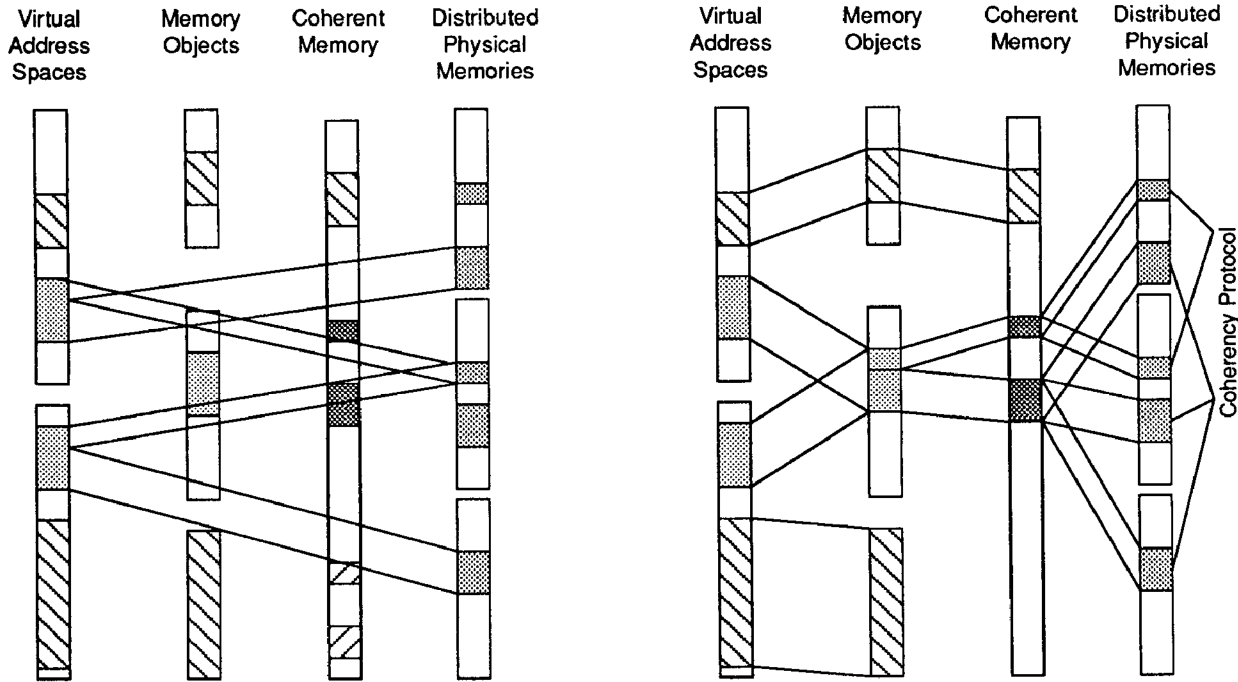

Figure 2: Logical and physical page mappings in PLATINUM. The coherent-to physical mappings for only one of the the memory objects are shown.

thread, generating a coherent memory fault. This circular dependence is broken by explicitly moving the kernel stack with the thread.

2.3 Implementation Structure

The coherent memory system consists of two modules:

1. The *coherent map* (Cmap) system is responsible for maintaining the coherency of the mappings from virtual to physical pages for each processor. The interface provided by the Cmap to the virtual memory system is similar to the Mach pmap interface.
2. The *coherent page* (Cpage) system is responsible for allocating and freeing coherent pages as well as the the physical pages that back them. It also maintains their coherency and implements a replication policy. The Cpage system includes the page fault handler and a *defrost daemon*.

The coherent and physical memory management systems use the following data structures (see Figure 3):

- The mappings from virtual addresses to memory objects and from memory objects to coherent pages are kept by the virtual memory system. For each address space the coherent memory system caches the composition of these mappings in a *Cmap*. A Cmap contains a table of virtual-to-coherent page mappings (Cmap entries), a queue of Cmap messages describing recent changes to the address space, a bit mask denoting processors with this address space active, and a separate local page table (Pmap) for each of these processors.

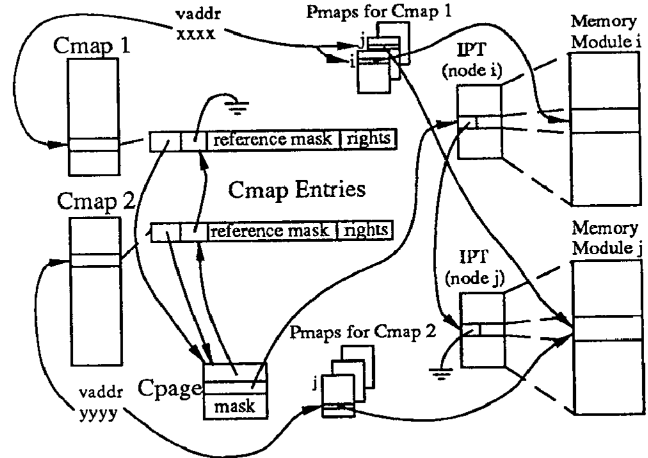

Figure 3: The data structures involved in the management of a Cpage mapped into two address spaces. The Cmap message queues are not shown.

- A *Cmap entry* is analogous to a page table entry. It contains a pointer to the coherent page, an access rights field, and a bit vector called the *reference mask*. If a processor has a virtual-to-physical translation for the coherent page in its Pmap, the bit corresponding to that processor is set.
- A *Cmap message* describes a change made to a virtual address space that affects virtual-to-physical mappings held by two or more processors. It contains a virtual address and a directive either to invalidate the current translation or to restrict the access rights in it. Each processor is responsible for making these changes before running any thread in that address space.
- The *Cpage table* is the list of all coherent pages. Each entry in the Cpage table describes the state of a Cpage. This information includes a directory of physical pages backing the Cpage and indicates whether there is a virtual-to-physical translation allowing write access. The directory consists of a bit mask indicating which memory modules contain a physical page backing the Cpage and a list of these physical pages. An entry also records the time of the most recent invalidation and whether the Cpage has been frozen by the replication policy.
- Each memory module contains an *inverted page table* describing the state of each physical page in the module. An entry indicates whether the physical page is allocated and to which coherent page.

3 Shared-Memory Coherency

The shared-memory coherency problem has two major facets, data coherency and address space coherency. Much of the literature on coherent caches for multiprocessors concerns the data coherency part of the problem. On UMA multiprocessors with coherent caches the address space coherency problem is primarily a matter of maintaining the consistency of address translation caches [5]. Given the lack of any direct hardware support for either form of coherency, PLATINUM solves both aspects of the problem in one unified framework; data coherency and address space coherency are implemented using a mechanism for invalidating or restricting the access granted by a mapping.

3.1 A NUMA Multiprocessor Shootdown Mechanism

When an address space is modified by the addition of new mappings or by relaxing the protection on a range of virtual addresses, it is easy to distribute the changes. Any processor attempting to use its expanded privilege will cause a bus error and thus be able to discover and react to the change. On the other hand, when an address space is restricted by removing mappings or restricting access rights, some additional mechanism is necessary to ensure consistency. For example, consider a UMA multiprocessor with a single shared page table per address space. Since page table entries are cached in the address translation cache (ATC) of each processor's hardware memory management unit, these cached copies must be invalidated whenever the corresponding page table entry is invalidated or restricted. Because address translation caches are usually private to the processor to which the MMU is attached, multiprocessor operating systems such as Mach use a software shootdown mechanism to implement this part of the address space coherency protocol [5]. The PLATINUM shootdown mechanism is very different from that used in Mach. The differences arise largely because the PLATINUM mechanism is designed specifically for NUMA multiprocessors.

Because code and data are replicated in PLATINUM each processor needs to have its own private set of virtual-to-physical mappings for each address space. While Mach uses a single shared page table (Pmap) per address space, each processor in PLATINUM must have its own private Pmap per address space. Since a Pmap is only a cache of the valid virtual-to-physical translations, it need not contain mappings for everything in an address space, rather only a working set for that processor. Thus, in contrast with a scheme examined by Holliday [15], scalability is not restricted by replication of page tables.

In addition to reducing latency and contention, using a local, private Pmap for each processor allows the construction of a fast shootdown mechanism. Black *et al.* discuss two problems that result from multiple processors sharing a single Pmap in Mach. If the processor initiating the shootdown instructs a target processor to flush its ATC before updating the Pmap, the target processor may reload an inconsistent entry. If, on the other hand, the initiating processor updates the Pmap before instructing the target processor to flush its ATC, the target processor may write back its ATC entry to update the reference or modify bits, thereby creating an inconsistent Pmap. Their solution to these problems is to stall the target processors while the initiator changes the Pmap. Since PLATINUM uses a Pmap per processor, it does not face either of these problems.

A consequence of the replication of mapping information is that the Pmaps must be kept coherent as well as the ATCs. Part of the protocol is performed by the processor initiating the shootdown and part is performed by the processors sharing the address space with the initiator. They communicate through the Cmap message queues and synchronize through interprocessor interrupts.

The initiating processor posts a short message describing the change of mapping to the Cmap message queue of each affected address space. A change to a specific address space affects only that address space, but a change of mappings required by the data coherency protocol must affect every address space in which the Cpage is mapped. Part of each message is the bit mask specifying the set of target processors that eventually have to apply the change to their Pmap for this address space. This set is exactly the set of processors appearing in the reference mask of each Cmap entry for this Cpage. The set of target processors is thus restricted to those that are actually using a mapping for this Cpage. Furthermore, a processor need only be interrupted to perform the change if the address space is currently active. The remainder of the target processors will update their Pmaps when they activate the address space. In contrast, the Mach shootdown mechanism must interrupt each processor with the address space activated, even if that processor has never referenced the page.

On the target processors the update is performed by a Cmap synchronization handler that is called as a result of an interprocessor interrupt or as part of the activation of an address space. Consequently, kernel code that runs at the interprocessor interrupt level or higher is not allowed to access coherent memory. The synchronization handler scans

the queue of change messages. If the processor appears in the target mask of a message, it applies the change to its Pmap and removes itself from the target mask. When the target mask is clear, the message is removed from the queue.

The memory management system obtains a significant reduction of overhead by deactivating the kernel address space when a processor begins running in user mode. This reduces the number of interprocessor interrupts each processor receives. When a processor reenters the kernel to service a trap or interrupt, it has to reactivate the kernel address space before it can access coherent memory.

3.2 The Data Coherency Protocol

The data coherency protocol is derived from a directory-based cache coherency protocol that uses selective invalidation of cache blocks [7]. When a processor tries to access a Cpage that has no local physical page backing it, the coherent memory mechanism can always choose either to make a local copy of a page or to create a mapping to an existing remote page. The ability to use remote mappings is especially important when multiple processors make frequent, interleaved, and fine-grain modifications to a shared data structure. The resulting interprocessor interference causes the frequent execution of any protocol to maintain coherence among multiple copies. By using remote mappings the mechanism can, in effect, selectively and dynamically disable replication and migration when interference is detected.

A coherent page can be in one of four states:

empty means that there are no physical pages backing the Cpage. Thus, there are no virtual-to-physical mappings to this page.

present1 means that there is exactly one physical page backing the Cpage and all virtual-to-physical mappings are restricted to read access. A virtual-to-coherent mapping may permit write access to the Cpage, but the virtual-to-physical mapping is restricted in order to implement the coherency protocol.

present+ means that there are two or more physical pages in different memory modules backing the Cpage. All virtual-to-physical mappings for the Cpage are restricted to read access. As above, a virtual-to-coherent mapping may permit write access to the Cpage.

modified means that there is one physical page backing the Cpage and at least one virtual-to-physical mapping allows write access.

Figure 4 is a transition diagram for the protocol. The **present1** state is distinguished from the **present+** state for performance reasons. The transition from **present+** to **modified** on a write miss requires the invalidation of at least one virtual-to-physical mapping and the reclamation of at least one physical page. The transition from **present1** to **modified** requires neither.

Transitions between states are triggered by page faults or the defrost daemon. When a page fault occurs during an attempted access to a non-**empty** Cpage, the Cpage system can either map an existing physical copy for remote access, or create and then map a local physical copy. For example, if there is a write miss on a Cpage in the **modified** state, the choice is between mapping the existing physical copy or allocating a local physical page, copying the data, and then invalidating the original copy. Similar decisions arise for the other cases.

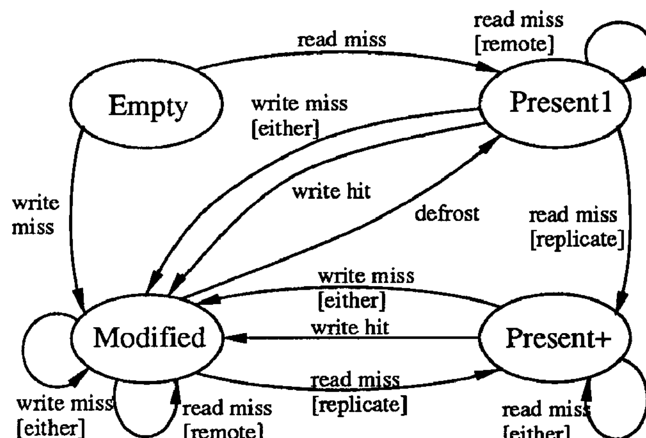

Figure 4: State Transition Diagram

A policy module within the Cpage system chooses the appropriate action on each page fault. The current policy uses the history of recent invalidations for the Cpage. Recent invalidation indicates that the Cpage is being actively write-shared. The Cpage system uses this information to limit the overhead of running the protocol by forcing remote mappings for recently invalidated pages and allowing replication for the others.

3.3 Replication and Data Coherency

Both the replication mechanism and the data coherency protocol are implemented by the page fault handler. When a page fault occurs, the Cpage fault handler searches the Cmap for an entry that maps the faulting virtual address. If an entry is found, the page fault is a coherent memory fault. Otherwise, the fault is passed to the virtual memory fault handler.

The Cmap entry contains a pointer to an entry in the Cpage table. The fault handler tests the bit mask in the Cpage to discover whether a local physical page backs it (see Figure 3). Since Cpages may be shared by multiple address spaces, a local physical copy may already exist. If a local copy exists, the handler applies a hash function to the index of the Cpage and scans the inverted page table to find the physical page. The inverted page table is used rather than the list of physical pages in the directory for the Cpage because the former is guaranteed to use strictly local memory accesses, thus decreasing both latency and potential contention. Even when contention is not a problem it is cheaper to scan over a few collisions in the inverted page table than to search the list of physical pages with remote

memory accesses.

If there is no local physical copy and the fault is a read miss, the fault handler consults the replication policy module to determine whether or not it should replicate the Cpage. If the Cpage is to be replicated, the handler uses the inverted page table to find a free physical page then allocates the physical page by entering the address of the Cpage in the inverted page table entry for the physical page. If the existing state of the Cpage is **modified**, the handler uses the shootdown mechanism to restrict all virtual-to-physical translations for the Cpage to read-only access. The handler then performs a block transfer from another physical copy, and adds the physical page to the directory.

Similar sequences of actions occur on a write miss. For example, if the state of the Cpage is **present+**, the handler first uses the shootdown mechanism to invalidate all virtual-to-physical translations for the remote physical copies, and then frees all of these pages. The handler concludes by mapping the chosen physical copy of the Cpage with the necessary access rights.

If the policy indicates that the Cpage should be *frozen* rather than replicated, there can only be one physical page backing the Cpage. Furthermore, the Cpage must be in a **modified** state. The handler creates a mapping for the remote physical page granting the full access rights permitted by the virtual memory system.

4 Performance and Choice of Replication Policy.

The copying of data in a PLATINUM page migration operation is a kernel-initiated, page-aligned block transfer of a known size. In the absence of contention this takes 1.11 ms for the default page size of 4K bytes.

The total time for a read miss that replicates a non-**modified** page ranges from 1.34 ms to 1.38 ms. The shorter time occurs when the relevant kernel data structures are local, while the longer time results from remote data access. Of this time, copying the page accounts for 1.11 ms, and the fixed overhead of allocating and mapping a physical page accounts for the remaining 0.23 ms to 0.27 ms.

A read miss that replicates a **modified** page takes from 1.38 ms to 1.59 ms if only one processor has to be interrupted to restrict its mapping to read-only access. The fixed overhead in this case ranges from 0.27 ms to 0.48 ms. The additional cost compared to a read miss on a non-**modified** is due to the address-space coherency protocol.

A write miss on a **present+** page takes from 0.25 ms to 0.45 ms when only one processor has be interrupted to invalidate its mapping and one physical page is freed. For up to 16 processors, the incremental delay to the initiating processor of interrupting each additional processor to invalidate a mapping and freeing a physical page is no more than 17 μs. Freeing a physical page uses one remote memory read and one write, accounting for about 10 μs of this time. We therefore believe that the incremental cost of interrupting a processor to restrict a mapping to be about 7 μs. In contrast, Black *et al.* report an incremental cost of 55 μs on a 16-processor NS32332 Encore Multimax [5].

These timings were gathered on a 16-processor BBN Butterfly Plus Multiprocessor. A processing node on this machine consists of a 16.67MHz MC68020 with a MC68851 MMU and 4 MBytes of physical memory.

4.1 When does it pay to migrate a page?

To decide when it is appropriate to migrate a data page rather than make a remote mapping, it is necessary to estimate the relative costs of each of these options. The following analysis is based on the contention-free latency of remote memory access. Contention, both at the memories and in the switch, increases latency by serializing requests. In the presence of contention the benefits of migration or replication can be much higher than indicated here.

Suppose a data structure, X, is shared and written by p processors; further suppose that X is the sole occupant of a coherent page. Each processor operates on X in a critical section as follows: obtain the lock for X, perform a computation f entailing r memory references on it, and release the lock. If this operation were encapsulated in a procedure call it might be performed in one of three ways:

- The operation is executed by the processor requesting it and the data is not moved. The operation is an ordinary procedure call using any combination of local and remote memory references to access the data.
- The data and the process executing the operation are co-located by moving the data. The operation is an ordinary procedure call using local memory references.
- The data and the process executing the operation are co-located by performing a remote procedure call. Access to the data uses local memory references.

While implementations of languages such as Emerald [17] on top of PLATINUM would utilize the third option, we restrict ourselves to consider the choice between the first two. Let C_{remote} be the cost of using remote memory references, C_{local} be the cost of using local references, and $C_{migrate}$ be the cost of moving the data. It is cheaper to move the data when

$$C_{remote} > g(p)C_{migrate} + C_{local}, \tag{1}$$

where $g(p)$ is the average number of data movements necessary to save a remote operation. It is the ratio of the total number of executions of f to the number of executions of f that would use remote memory access if the data were not moved. When p processors access X in strict round-robin order, $g(p) = p/(p-1)$. For example, consider two processors that alternate in touching X. If X is not moved, there will be one remote and one local execution of f per cycle. If it is moved, there will be two local executions of f and two data movements per cycle. Thus $g(2) = 2$. This is the worst-case scenario. For large p, $g(p)$ approaches 1. If the operations are not interleaved among the processors then $g(p)$ can be less than 1.

Let

s be the size of a page expressed in terms of the typical unit of access. On the Butterfly Plus this is a 32-bit word.

T_l be the time to perform a typical local memory reference on a 32-bit word. On a Butterfly Plus this is about 320 ns.

T_r be the time to perform the corresponding remote memory reference. On the Butterfly Plus this is about 5000 ns to read a 32-bit quantity. Write operations are faster.

ρ	S_{min}, minimum page size in words $g(p) = 0.5$	$g(p) = 1$	$g(p) = 2$
0.17	1070	never	never
0.24	445	never	never
0.35	232	973	never
0.48	149	435	never
0.60	111	298	1784
0.75	85	210	793
1.0	61	141	412
1.5	39	84	210
2.0	28	61	141

Table 1: Inequality 2 evaluated at some interesting points. It always pays to migrate data when the page size is greater than S_{min}.

T_b be the time to copy a word in a page migration operation. This is about 1100 ns on the Butterfly Plus.

Define $\rho \equiv r/s$ to be the density of references to X. For example, if the size of X is s and f reads and writes every location in X, $\rho = 2$. On the other hand, if X occupies only half the page and f writes one half of X's data, $\rho = 0.25$.

We therefore have $C_{local} = \rho s T_l$, and $C_{remote} = \rho s T_r$. The cost of migration is divided into the cost of block transfer, sT_b, plus a fixed overhead, about 0.48 ms in the current implementation. Substituting these into inequality 1 and rearranging the terms, we conclude that it always pays to migrate when

$$s > \frac{107g(p)}{\rho - 0.24g(p)}. \quad (2)$$

Note that the constant in the numerator is proportional to the fixed overhead of the migration operation and that the coefficient of $g(p)$ in the denominator is the ratio $T_b/(T_r - T_l)$. From this we make the following observations:

- To determine when migration is economical, the ratio $T_b/(T_r - T_l)$ of block transfer time to the time that can be saved by using local rather than remote memory access operations is the single most important characteristic of the architecture. It puts a lower bound on the minimum reference density ρ for which migration makes sense for any block size. This in turn bounds the minimum usable page size. *The existence of a fast block transfer mechanism is vital to the performance of any program that uses data migration and replication on a NUMA machine!*
- For each $g(p)$ and ρ pair, the need to amortize the fixed overhead of the coherence protocol puts a lower bound on the page size that can be used economically. For a fixed $g(p)$ and ρ, a decrease in overhead results in a proportional decrease in the minimum page size for which migration makes sense.
- With round-robin access, as the number of processors sharing X increases, $g(p)$ decreases towards 1, thus making migration more attractive.

These factors determine the granularity of data access that must be seen in the application to ensure that migration is always the correct action for a given page size. Some values for inequality 2 are presented in Table 1.

This analysis emphasizes the importance of coarse data-access granularity for attaining good performance on a NUMA machine, especially if data migration on a per-page basis is used. A large page size allows us to better amortize the fixed overhead of data transfer and thus to tolerate a slightly smaller ρ. For a fixed granularity of data access smaller than the size of a page, however, ρ is inversely proportional to page size, thus negating any potential advantage of increasing page size. On the other hand, if a program has a granularity of data access that is greater than the size of a page, ρ remains more or less constant as pages grow. For a fixed problem size one would expect the granularity of sharing to decrease as the number of processors in a multiprocessor increases. On the other hand, we believe, as do others [28, 14], that a major role of parallel machines is to solve ever-larger problems rather than to solve fixed-size problems in ever-shorter times. These larger problems will allow the continued use of coarse granularity as systems are made larger.

4.2 Replication Policy

While the replication and migration of data has significant benefit when data access granularity is relatively coarse, the overhead of trying to maintain coherency in the presence of fine-grained write-sharing could be prohibitively expensive. In such circumstances it is less expensive to access remote memory than to try to migrate or replicate the data. Since the choice between data movement and remote access depends upon the relative costs of the alternatives, we have delayed discussion of the replication policy until the details of the mechanism and of its cost have been presented. PLATINUM is designed to support experimentation with a family of policies. We focus on the interim policy currently used.

Since invalidations occur as a result of interprocessor interference, all policies use a recorded history of recent invalidations to estimate the interference for each coherent page. The current version uses a minimal history consisting of a timestamp for the most recent invalidation by the coherency protocol of a mapping for that coherent page. On a page fault handled by coherent memory, a coherent page is replicated or migrated if the the last invalidation by the protocol was at least t_1 in the past. Otherwise it is frozen rather than replicated. Since invalidations cause the Cpage to go into the **modified** state and since it could not have been replicated since then, there can only be one physical page backing a frozen Cpage. While it remains frozen, all new mappings to a Cpage are to that single physical page. We have used two policies for dealing with faults occurring after the t_1 ms period expires on a frozen Cpage. The default policy is to continue to create remote mappings for the Cpage until the page is explicitly thawed. The alternative is to allow the frozen coherent page to be replicated and thus thawed as a consequence of an attempted access. The programs we have examined thus far exhibit no significant difference in performance between these policies.

Based on the speed of the Butterfly processor and the need to amortize the replication of a coherent page over a reasonable number of accesses, t_1 is currently set to 10 ms. A few tests indicated that application performance is insensitive to varying t_1 from 10 ms up to about 100 ms. Once the collection of application programs has grown to a

reasonable size we will perform systematic experiments on the effects of varying this and other parameters.

After all of the threads that share a frozen Cpage have mappings to it, further access to that Cpage causes neither additional faults nor the associated overhead. Since the coherency protocol as described thus far is driven strictly by page faults, the Cpage could remain frozen permanently. While it may be appropriate to freeze a Cpage at a particular point in the execution of a program, a change in the access pattern of that page may make it desirable to thaw it in the future. PLATINUM therefore has a simple mechanism for thawing pages, thus allowing the memory management system to react to phase changes as well to thaw any incorrectly frozen pages.

The Cpage module maintains a list of frozen Cpages and a clock interrupt every t_2 seconds activates the *defrost daemon* to invalidate all mappings to the frozen pages. Subsequent access attempts will cause faults that may replicate or migrate a recently thawed coherent page. To keep the overhead low, t_2 is currently set to 1 second. Reducing t_2 may allow coherent pages frozen accidentally to be replicated sooner, but it just adds overhead for coherent pages that should remain frozen.

An alternative is to maintain the list of frozen pages as a priority queue ordered by thaw time. This allows the daemon to run more often than every t_2 seconds. It also allow t_2 to be set adaptively on a per-page basis. Although there is evidence that thawing frozen pages is important for performance, we do not yet have reason to believe that a more sophisticated policy for thawing will have much effect. Since a more sophisticated policy would add overhead to the system, we plan to continue to use the simple policy described above until the problem is better understood.

A possible reason for the access pattern of a page to change is that two or more variables with different access patterns are in that page. For example, co-locating a synchronization variable such as a lock or event count with a read-only variable on one page can lead to problems because they demand very different treatments from the memory management system. Active use of synchronization variables will cause their pages to be frozen while a read-only variable should be replicated. The preferred solution to this problem is for the programmer, the compiler, and the language run-time support to be intelligent about the allocation of variables to virtual pages. Even if this allocation is done poorly, thawing can salvage reasonable performance if each variable is used primarily in a different phase of the program.

Experiences with our first version of the Gaussian elimination program, described in the next section, provide anecdotal evidence of the importance of intelligent memory allocation, thawing, and performance instrumentation. The program takes the problem matrix size as a parameter and writes this value to a variable during the startup phase. The matrix size is used in the termination test of the inner loop of the algorithm so it is vital that each processor have a local copy. The slave threads did not make private copies of this variable, but the page was replicated. Later we added a spin-lock variable to facilitate measurement of execution times. It is used as a barrier at the start of the elimination phase of the program and is not touched thereafter. Spinning on the lock froze the Cpage. Consequently, all but one thread generated a remote access in its inner loop. This increased the latency for accessing the shared variable. This dramatically increased the execution time and became a bottleneck with five or more processors.

In addition to timing data, the kernel produces a detailed report on the behavior of memory management. For each Cpage this includes the number of coherent memory faults, a measure of contention in the Cpage fault handler for that page, and whether the Cpage was frozen by the replication policy. Given this instrumentation it was a simple matter to diagnose the problem and program around it by giving each thread a private matrix-size variable. Thawing was soon added to the kernel and the old version of the program took less than two seconds more to run than the new version. The overhead of running the defrost daemon adds no measurable overhead to the new version of the program.

5 Application Performance

We report preliminary performance measurements for three application programs running on PLATINUM. Each of these programs has a memory access pattern distinct from the others.

5.1 Gaussian Elimination

The first application we examined was the simulation of Gaussian elimination described in the introduction. This particular computation was chosen because it had been studied previously on an earlier version of the Butterfly for a variety of programming systems and styles [18, 19]. LeBlanc compared the performance of an implementation on the Uniform System from BBN [3] with Gaussian elimination implemented on SMP [20], a message passing library developed at the University of Rochester. We used the same 800x800 matrix as LeBlanc.

The PLATINUM implementation is similar to the coarse-grain implementation on the Uniform System found to be the most efficient in LeBlanc's study. There is a single thread per processor and each thread is statically allocated a number of rows of the matrix. In each round some thread selects a pivot row which is then read by all of the other threads. Each thread then performs the elimination operation on its set of rows.

The differences between the two versions of the Butterfly reduce the accuracy of quantitative comparisons of performance measures. Nevertheless, such measures provide a framework for qualitative comparison. The program running on PLATINUM yields a 16-processor speedup of 13.5 versus 10.6 for the Uniform System program [18]. In contrast, the SMP message-passing implementation yielded a speedup of 15.3.

An examination of the post-mortem statistics gathered by the kernel shows that the PLATINUM implementation exhibits high contention in the Cpage fault handler for Cpages that contain the pivot rows. This is attributable to a serialization in hardware of the replication of the data backing a Cpage. As expected, only the Cpage containing an array of event counts used for synchronization was frozen.

5.2 Merge Sort

This is a parallel merge sort using a simple tree of merge operations, each of which is performed by a single thread. We chose this program because it had been studied on a

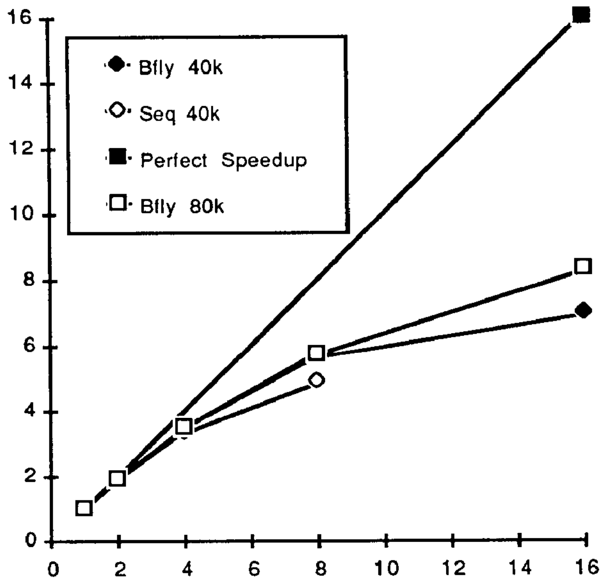

Figure 5: Merge Sort Speedup

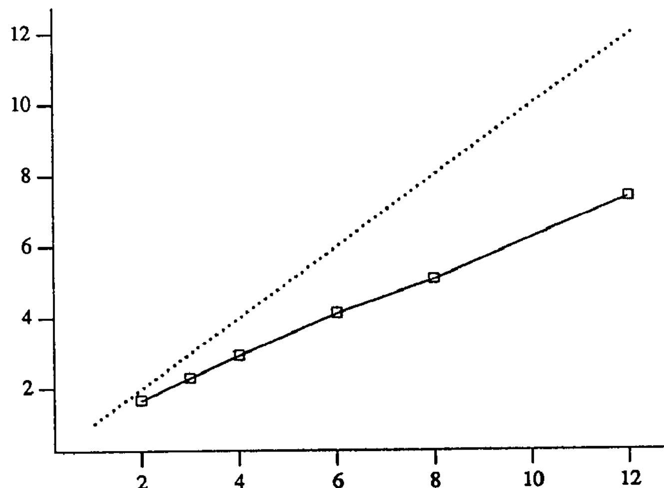

Figure 6: Recurrent Backpropagation Simulator (Speedup vs. Processors)

Sequent Symmetry Multiprocessor [1]. The Sequent Symmetry is a UMA multiprocessor. The one used in the study had model A processors with 8Kbyte write-through caches.

Figure 5 shows the measured speedup curves for this program. The program shows better speedup running on the Butterfly Plus under PLATINUM than on the Sequent Symmetry for the same size problem on the same number of processors. We believe this is due to the small cache size and write-through policy on the Sequent. During each merge phase one half of the data to be merged will already be in the merging processor's local memory. Furthermore, with the linear access pattern of merging, the processor will touch all of the data prefetched by each coherent page fault. The problem is large enough, however, that none of the data will remain in the Sequent cache between merge phases.

5.3 Neural Network Simulator

A very different application is a simulator used by neural network researchers at the University of Rochester studying recurrent backpropagation networks [27]. Unlike the others, this program was developed by someone with no previous experience programming the Butterfly Plus. While the other programs were written to exploit coarse-grain parallelism on large amounts of data, the simulator operates on much less data and at a very fine granularity.

We measured (Figure 6) the performance of a simulation of a three layer network learning a classic encoder problem [24]. There were 40 units and 16 pairs of inputs and outputs. The simulator is parallelized by simple for-loop parallelization on units. Each processor continually simulates a set of units depending only on the atomicity of memory operations for synchronization when it accesses data shared with other threads. The non-determinism produced by the lack of synchronization introduces negligible variability of execution time. Given the very fine-grain nature of the algorithm, PLATINUM cannot use replication or migration to good advantage. The coherent memory system quickly gives up and the data pages of the application are frozen in place. The speedup curve is linear over the range measured, but the extensive use of remote accesses limits the contribution of each incremental processor to about 1/2 that of a processor that makes only local memory references.

6 Experiences Programming on a Coherent Memory

In our experience, it is much easier to write applications to run on coherent memory than to run on non-uniform physical memory. PLATINUM programs are smaller than both Uniform System programs and programs using message-passing styles because one need not write code either for explicit communication, or for explicit management of data location. For example, the code for the elimination phase of the PLATINUM, Uniform System, and SMP implementations of Gaussian elimination are 17, 41, and 64 lines long, respectively.

Despite the apparent familiarity of PLATINUM's abstract machine model, a programmer still needs to understand and apply certain fundamental facts about parallel programming on a NUMA machine. It is of overwhelming importance to avoid programming styles entailing fine-grain write-sharing. Whether memory is being managed automatically by the coherent memory system or explicitly by the programmer, this fine-grain write-sharing introduces both latency that reduces the effective processor speed and memory contention that serializes logically parallel computations. It is vital that most of the sharing of writable data be done at coarse

enough spatial and temporal granularities that a fast block transfer mechanism can be used effectively.

In order for the coherent memory system to effectively manage data location, the programmer or compiler must understand the sharing properties of data. Data with different access patterns should not be co-located on a single page. The private data of each thread should be separated from private data of other threads and from shared data. Read-only data should be kept separate from modifiable data. Coarse-grain modifiable data should be separated from fine-grain modifiable data such as locks. A run-time library for defining disjoint memory allocation zones and for specifying page-aligned allocation helps PLATINUM programmers to do this with a minimum of effort, even without compiler support. Because a typical NUMA multiprocessor has a very large physical memory, the internal fragmentation introduced by this strategy has little impact and is vastly preferable to interprocessor interference.

7 Architectural Considerations

The benefits of replication cannot be measured solely in terms of the ratio of local to remote memory access times. As the degree of parallelism increases on a machine with a large number of processors, contention for memory modules and for the interconnection network become the dominant factors determining performance. The most important impact of coherent memory is that it effectively uses local memories as caches to reduce contention.

An effective block transfer mechanism is critical to an efficient implementation of coherent memory. It should be both fast and asynchronous with respect to program execution. The analysis in Section 4.1 quantifies the importance of block transfer speed in one scenario. Although the Butterfly Plus has a fast, asynchronous block transfer mechanism, it consumes 75% of the available local memory bus bandwidth on both nodes involved in the transfer. Both processors are memory-starved during a block transfer. Redesigning the memory system to allow more concurrency between processing and block transfers would help to reduce further the effects of memory contention.

Although the Butterfly Plus does not have data caches in the processor nodes, the PLATINUM coherent memory system is compatible with a generation of NUMA multiprocessors with local caches but without internode coherency support. In addition to reducing latency on local memory operations, local data caches would reduce contention for the local memory module between the local processor and remote memory operations. Such local data caches could be relatively cheap because they need not incorporate a hardware cache coherency mechanism. Cache coherency would be maintained by the coherent memory system. Almost all data is cachable. Only modified Cpages that are mapped by remote processors cannot be cached. Replicating a modified Cpage would, however, require flushing a write-back cache, slowing the invalidation operation.

8 Related Work

The management of NUMA memory is a topic of considerable current interest. Recent studies of methods for managing the location of data in a NUMA machine include the analysis and simulation of competitively optimal NUMA memory management by Black *et al.* [4], Scheurich and DuBois' simulation of data migration in mesh-connected NUMA machines [25], and Holliday's simulation of data migration on a Butterfly [16]. The design of the Psyche memory manager [21] contains a layer that deals with NUMA data location issues.

In Bolosky's addition of NUMA memory management to Mach on the IBM ACE Multiprocessor Workstation [6] writable pages are never replicated and are allowed to migrate only a small number of times before being frozen in global shared memory. While this results in performance improvements compared to static placement, our experiences with coarse-grained sharing indicate that there is room for improvement. If write operations on a large piece of a data structure are not interleaved at a fine grain, it continues to be appropriate to migrate data throughout the lifetime of an application. Further it is not only appropriate to replicate immutable objects, but also those modifiable objects that are, either by accident or design, not modified during some phase of execution.

While one effect of replication and migration in the PLATINUM coherent memory system is the reduction of latency, we contend that for large hardware configurations a far more important benefit is the reduction of memory and switch contention. Therefore, we have have not expended much effort trying to tune the mechanism for the optimal placement of frozen pages that are being actively modified at a fine granularity by multiple processors. While careful placement and migration can reduce average access latency in the absence of contention, there is no demonstrated reduction in contention. Since the proposed placement mechanisms are not cheap, entailing hardware reference counts [4, 25] or simulations of reference counting in software [16, 21], we believe that it is better to have a simple, low-overhead placement policy and to devote more resources to reducing contention by reducing the amount of fine-grain write-sharing.

9 Status and Future Directions

Our experiences thus far indicate that the PLATINUM memory management system will achieve its goals. Foremost, the memory management system makes it easier to program a NUMA architecture without an unacceptable sacrifice in performance. Although initial programming experiments used the kernel interface directly without too much programmer effort, we are rapidly accumulating run-time libraries, shells, and other support software to further ease the programming process. An important part of this will be the installation of instrumentation for performance monitoring, analysis, and visualization [12]. The feedback from such instrumentation is useful to application programmers, compiler writers, and system implementors for NUMA machines.

We are continuing to study the behavior of the coherent memory system under a variety of applications. Once the collection of applications has grown to a reasonable size we will systematically experiment with the implementation by changing parameters such as page size and replication policy.

The kernel itself is designed to scale well to machines with a much larger number of processors. Its decentralized design keeps the number of remote memory accesses in the kernel to a minimum. We are particularly pleased with the success of the decentralized and concurrent implementation of the

coherency protocol, especially the low incremental cost per shootdown and the techniques for reducing the number of processors involved in a shootdown.

Although providing coherent memory transparently in the operating system has proven itself useful, it is not hard to construct scenarios in which better performance could be obtained if interface between the application and the memory management system were not so transparent. The kernel interface will be extended to support these. While such information could be provided by the programmer directly, this additional burden runs contrary to the goal of providing a simple programming environment. We therefore anticipate that these hooks will be utilized primarily by programming languages and their run-time support.

In its current incarnation, PLATINUM is a limited experimental platform for experimenting with the implementation of coherent memory. We will extend it as necessary to serve this purpose. On the other hand, dealing with issues such as file systems and protection is not in our plans. When and if it becomes appropriate to make coherent memory available in a general-purpose operating system, we anticipate reintegrating those parts of PLATINUM with Mach.

Acknowledgements.

We thank the referees and Hugh Lauer for their constructive comments. Niki Fowler deserves special credit for her editorial assistance on the revised version of this paper.

References

[1] R. J. Anderson. An experimental study of parallel merge sort. Technical Report 88-05-01, Department of Computer Science, University of Washington, May 1988.

[2] James Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors.* PhD thesis, Department of Computer Science, University of Washington, February 1987.

[3] BBN Laboratories, Cambridge, Massachusetts. *The Uniform System Approach To Programming the Butterfly Parallel Processor*, October 1985.

[4] D. Black, A. Gupta, and W.D. Weber. Competitive management of distributed shared memory. In *Spring Compcon*, 1989.

[5] D.L. Black, R.F. Rashid, D.B. Golub, C.R. Hill, and R.V. Baron. Translation lookaside buffer consistency: A software approach. Technical Report CMU-CS-88-201, Department of Computer Science, Carnegie-Mellon University, December 1988.

[6] William J. Bolosky, Michael L. Scott, and Robert P. Fitzgerald. Simple but effective techniques for NUMA memory management. Technical report, Department of Computer Science, University of Rochester, March 1989.

[7] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[8] D. Cheriton, A. Gupta, P. Boyle, and Hendrik Goosen. The VMP Multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 410–421, June 1988.

[9] D. Cheriton, G. Slavenburg, and P. Boyle. Software-controlled caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 366–374, June 1986.

[10] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node Butterfly Parallel Processor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531–540, August 1985.

[11] R. Fowler and A. Cox. An overview of PLATINUM: A platform for investigating non-uniform memory. Technical Report TR-262, Computer Science Department, University of Rochester, November 1988.

[12] R.J. Fowler, T.J. LeBlanc, and J.M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distribute Debugging*, pages 74–182, Madison, Wisconsin, May 1988. Association for Computing Machinery.

[13] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.

[14] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1989.

[15] Mark A. Holliday. Page table management in local/remote architectures. Technical Report CS-1988-2, Department of Computer Science, Duke University, July 1988.

[16] Mark A. Holliday. Reference history, page size, and migration daemons in local/remote architectures. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 104–112, April 1989.

[17] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, pages 109–133, February 1988.

[18] T. J. LeBlanc. Shared memory versus message-passing in a tightly-coupled multiprocessor:a case study. Technical Report Butterfly Project Report 3, Computer Science Department, University of Rochester, January 1986. A shorter version appears in Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 463–466.

[19] Thomas J. LeBlanc. Problem decomposition and communication tradeoffs in a shared-memory multiprocessor. In Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures*, volume 13 of *The IMA Volumes in Mathematics and its Applications*, pages 145–163. Springer-Verlag, 1988.

[20] Thomas J. LeBlanc. Structured Message Passing on a shared-memory multiprocessor. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 188–194, January 1988.

[21] Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Memory management for large-scale NUMA multiprocessors. Technical report, Department of Computer Science, University of Rochester, March 1989.

[22] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986.

[23] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Parallel Distributed Processing,* chapter Learning internal representations by error propagation, pages 318–364. MIT Press, 1986.

[25] C. Scheurich and M. DuBois. Dynamic page migration in multiprocessors with distributed global memory. *IEEE Transactions on Computers*, 38(8):1154–1163, August 1989.

[26] Michael L. Scott and Alan L. Cox. An empirical study of message-passing overhead. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 242–249, September 1987.

[27] P. Simard, M. Ottaway, and D. Ballard. Analysis of recurrent backpropagation. In *Connectionist Summer School Proceedings*, 1988.

[28] L. Snyder. Type architectures, shared memory, and the corollary of modest potential. In *Annual Review of Computer Science*, volume 1, pages 298–317. Annual Reviews Inc., 1986.