(4)

An Overview of PLATINUM
A Platform for Investigating Non-Uniform Memory (Preliminary Version)

Robert J. Fowler and Alan L. Cox

Technical Report 262
November 1988

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 262	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) An Overview of PLATINUM, A Platform for Investigating Non-Uniform Memory (Preliminary Version)		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert J. Fowler and Alan L. Cox		8. CONTRACT OR GRANT NUMBER(s) N000014-84-K-0655 DACA 76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department 734 Computer Studies Bldg University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS D. Adv. Res. Proj. Agency 1400 Wilson Blvd. Arlington, VA 22217		12. REPORT DATE November 1988 13. NUMBER OF PAGES 20
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Res. Information Systems Arlington, VA 22217 U.S. Army ETL Fort Belvoir, VA 22060		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Multiprocessor, operating system, memory management, NUMA, caching, cache coherency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

PLATINUM is an experimental operating system kernel designed to facilitate research on memory management systems for Non-Uniform Memory Access (NUMA) Multiprocessor Architectures. It exports to user programs a simple abstraction of a shared memory multiprocessor in which all memory appears to be uniformly and rapidly accessible from all processors in the machine. The perceived uniformity on top of a non-uniform physical memory architecture is supported by an abstraction called coherent memory. Implemented in software as an extension

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

of a directory-based caching mechanism using invalidation, coherent memory attempts to transparently migrate and replicate data to locations that are physically close to the processors that use it. A fundamental property of PLATINUM coherent memory is that it automatically reverts to the use of remote memory access for data that is not amenable to caching. PLATINUM currently runs on BBN Butterfly Plus Computers.

This report is an overview of PLATINUM. In addition to motivating the project and presenting our research plans, we describe the interface that the kernel provides to its users.

An Overview of PLATINUM
A Platform for Investigating Non-Uniform Memory
(Preliminary Version)

Robert J. Fowler
Alan L. Cox

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 262

November 1988

Abstract

PLATINUM is an experimental operating system kernel designed to facilitate research on memory management systems for *Non-Uniform Memory Access* (NUMA) Multiprocessor Architectures. It exports to user programs a simple abstraction of a shared memory multiprocessor in which all memory appears to be uniformly and rapidly accessible from all processors in the machine. The perceived uniformity on top of a non-uniform physical memory architecture is supported by an abstraction called *coherent memory*. Implemented in software as an extension of a directory-based caching mechanism using invalidation, coherent memory attempts to transparently migrate and replicate data to locations that are physically close to the processors that use it. A fundamental property of PLATINUM coherent memory is that it automatically reverts to the use of remote memory access for data that is not amenable to caching. PLATINUM currently runs on BBN Butterfly Plus Computers.

This report is an overview of PLATINUM. In addition to motivating the project and presenting our research plans, we describe the interface that the kernel provides to its users.

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.

1 Introduction.

PLATINUM is an experimental operating system kernel designed to facilitate research on memory management systems for *Non-Uniform Memory Access* (NUMA) multiprocessor architectures. The name "PLATINUM" is an acronym for "Platform for Investigating Non-Uniform Memory". Specifically, it provides a platform for evaluation implementations in software of *coherent memory* abstractions on top of non-uniform physical memory architectures. The distributed, shareable memory of a NUMA 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. An implementation of coherent memory within PLATINUM can replicate and migrate data to locations close to the processors using that data, thus creating the appearance that memory is uniformly and rapidly accessible. The protocol for controlling this data movement is an extension of a directory-based caching algorithm using selective invalidation to maintain coherency [1, 4]. Because PLATINUM runs on a NUMA machine, a coherent memory protocol always has the option of choosing not to replicate or migrate data, but to use the underlying remote access mechanism instead, in effect dynamically disabling caching on a block-by-block basis. This is crucial because the modification of shared data at fine temporal and spatial granularities can cause interference that costs more than not having caching at all. This effect can be especially expensive with the large block sizes associated with software-assisted caching.

Trace-driven simulation is a standard technique for evaluating multiprocessor memory management protocols. We considered simulating coherent memory protocols prior to implementing them in PLATINUM, but we rejected this approach because we believe that simulations would be able to produce unequivocal results only with a level of effort and computer time considerably greater than the straightforward strategy of implementing a small, instrumented multiprocessor kernel, writing (or converting) a test suite of application programs to run on that kernel, and observing the instrumented system in operation. Factors that influenced our decision to implement rather than simulate included:

- Obtaining reference traces can be a major effort. Existing traces recorded by computer manufacturers are not in the public domain and can be difficult to obtain. Recording accurate multiprocessor traces is especially expensive and slow because it entails the creation of a serialized recording of the memory references of all of the processors in the computation. In a set of traces that has obtained recent prominence in the literature [1, 2, 12, 26] each run is able to record only a few seconds of references for four processors before consuming the available buffer space. In contrast, the execution of an instrumented kernel can be observed for long periods of time on larger numbers of processors. Furthermore, the relative ease of actually running programs means that many more experiments can be performed.

- Trace driven simulations are useful for answering questions about how an existing program would run on the simulated system. Our experience has been that because performance[1] is the driving force behind parallel computation, programmers and compilers optimize parallel programs for the architectures on which the programs are run.

[1] We are not considering fault-tolerant systems.

Running and tuning applications on an instrumented kernel has the potential to yield insights not obtainable by simulation with traces generated by a "dusty deck" optimized for some other parallel architecture.

- The acid test of any idea is its implementation. Running an implementation on real hardware forced us to address all the details of building a memory manager.

This paper is an overview of PLATINUM. It presents a rationale for the project, describes the interface that the kernel exports to user programs, enumerates some of the experiments we anticipate performing, and points to future directions. A companion paper [13] describes the details of the design and implementation of the memory management subsystem.

PLATINUM currently runs on BBN Butterfly Plus Computers [9]. We are in the process of implementing the first real application: a neural network simulator for experimenting with asynchronous recurrent back-propagation [25].

2 NUMA shared memory and its management.

Any large computing system must have an internal structure that is both physically and logically distributed, composed of modules that communicate through some form of interconnection network. This network introduces limitations on the speed of the system: physical propagation delay, switching delay, arbitration protocols that must be executed if there is potential contention, and the serialization that occurs when contention actually exists. These factors make large memories slower than small ones and make memory systems that can be shared among multiple processors and DMA channels slower than those accessible from only one device. The design of the memory systems of all large fast computers must address these factors in order to keep average access times small enough to allow the central processor to run at close to full speed. The universal approach to this problem is to construct a memory hierarchy that contains fast local memories at strategic points in the system. In some cases, such as general-purpose scalar and vector register sets, these memories are explicitly managed by user programs. The architecture can also provide implicit management, as in the cases of data, instruction, and address translation caches.

The problem of memory speed in a complex system is exacerbated when multiple processors share some of the memory system. Because parts of the memory system are necessarily far from some of the processors, if physical memory speed is to be uniform (as in the case of so-called "dance hall architectures" [27]) then it must be uniformly slow. As in the uniprocessor case, the most common method of increasing average memory speed is to add fast local memories, ranging from hardware caches to user-managed general-purpose registers and private memories. The design of the IBM RP3 [21] multiprocessor contains provisions for all of these.

Adopting the terminology of [30] we refer to multiprocessors in which all local memory (except for processor registers) is implicitly managed by the architecture as uniform memory access (UMA) machines. Memory locality is transparent to user programs. Although it may be possible to detect and induce some variability of access times due to the behavior of the cache, the architecture is designed to maintain the illusion of a single fast global memory.

The cache controller executes a *coherency protocol* [10], ensuring that all processors have a consistent view of memory when multiple caches share a data item. The economy of implementing coherency protocols by "snooping" on a shared memory bus has enabled a whole generation of modestly sized multiprocessors. These protocols increase effective memory speed seen by the processors and reduce the rate at which each processor uses the bus to access main memory, thereby allowing more processors to be added to the system. Although these protocols are effective with a modest amount of parallelism, the scaleability of these strategies is limited by bus bandwidth, which is consumed by memory traffic and by a rapidly increasing number of bus cycles used to maintain coherency [3, 4]. Directory-based caching schemes are more scaleable than bus-based schemes [2] and are amenable to multi-stage memory-switch architectures. Implemented in hardware, they could be the basis for the design of more scaleable UMA machines.

An alternative to building uniformity into the architecture is to build the multiprocessor so that the physical location of data in memory is both apparent and controllable by the user program. Locality remains transparent in the sense that a single mechanism suffices to access local memory, shared global memory, or the local memory of another processor; however, the time for a processor to access physical memory does vary with its location. We refer to these as non-uniform memory access (NUMA) architectures. NUMA architectures were among the earliest large multiprocessors [29, 15] and they continue to be prominent [7, 9, 21]. NUMA architectures are more scaleable than UMA architectures because they do not depend on coherent hardware caches, instead displacing the management of the memory hierarchy from the architecture and forcing it into higher layers of the system.

The physical placement of code and data is critical to performance on NUMA computers. For example, on the BBN Butterfly no program can afford to fetch its instructions from remote memory. Therefore, shared code must be replicated among all the local memories of processors that use it. Furthermore, considerable effort can be expended on situating concurrently accessed data to reduce both access time and contention. Methods include locating data at the processor that accesses it and scattering the data among the memory modules. Even when a programming system manages location automatically, slow memory access is still apparent to programmers. For example, the Uniform System [8] package on the Butterfly transparently scatters arrays of globally shared data among the memory modules of the machine. This has the effect of statistically reducing contention and making access times more uniform but noticeably longer than those for local memory. In reaction to this, application programmers concerned with performance adopt the programming idiom of locking a piece of the shared data, using a fast block transfer to copy the data to a memory location known to be local, operating on it there, and then block-copying it back to its original location [8, 17].

Explicit management of the physical location of code and data by users is a burden on the programmer and can be difficult even in scientific applications that use very regular patterns of sharing. Because the choice of data structures and algorithms affects the viability of each location strategy, this management of location is often done *ad hoc* for each application, consuming a significant amount of programmer effort. A premise of this project is that a programmer should be able to concentrate on the application and spend less time programming the architecture. The implicit management of memory hierarchies

on uniprocessors makes the programmer's task for most applications much easier at the cost of acceptable sacrifices in performance. The implicit memory management of PLATINUM is intended to achieve a similar state of affairs on NUMA multiprocessors.

There are, however, some programs whose behavior defeats the purpose of implicitly managed memory hierarchies. For example, programs exhibiting poor locality of reference can induce thrashing in paging systems. A critical advantage of implicit NUMA memory management in PLATINUM over schemes such as the software caching of Li's Distributed Virtual Memory [19] and the software-controlled caching of the VMP Multiprocessor [11, 12] is the option of using remote memory references. Interference causes poor cache performance when memory is shared at a very fine temporal grain by multiple processors. The effects of this interference are magnified when several shared data items are packed into a single cache block. This effect can be severe for implementations of caching in software because they have a relatively high fixed cost per operation that is amortized by using a relatively large block size. With the option of performing remote references, in effect dynamically disabling caching when interference is detected, PLATINUM has the potential to accommodate fine-grained sharing without suffering the overhead of cache interference. Although remote memory access is more expensive than local access, it is still much cheaper than communicating through message passing for fine-grain communication and synchronization.

We are not alone in the investigation of NUMA memory management. Recent work in the area includes the analytic studies of optimal NUMA memory management by Black *et al.* [5], Sheurich and Dubois' study of the benefits of data migration in mesh-connected NUMA machines [24], and Bolosky's addition of NUMA memory management to Mach on the IBM ACE Multiprocessor Workstation [6].

Section 3, below, discusses the abstract machine model implemented by PLATINUM. Sections 4 through 7 are a detailed exposition of the implementation of the model by the interface the kernel exports to its users. Sections 8 and 9 present our plan for future research using PLATINUM and review the current status of the project. For more detail on the motivation, design, and implementation of PLATINUM, and especially of the coherent memory layer, see [13].

3 The PLATINUM Model.

The model of computation that PLATINUM exports to user programs is a virtual UMA multiprocessor architecture in which all primary memory accessible to user programs appears to be in an abstraction of a fast (on average) shared physical memory module that is uniformly accessible from all of the processors in the system. The physical location of data in primary memory is hidden from the user. As stated above, the kernel implements this abstraction on top of the underlying NUMA architecture through the use of software caching techniques built into its memory management system.

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 virtual address range of the same size, subject to hardware alignment restrictions. Neither the virtual address range nor the access rights need be the same in every address space. Since they have global names, memory objects serve as the unit of data- or code-sharing between address spaces.

A *thread* is a kernel-schedulable 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 spread among multiple processors.

A *Port* is a protected message queue that can have any number of senders and receivers. Messages are variable-length arrays of zero or more bytes. Globally named, ports provide a communication medium usable by threads that do not share access to a common memory object. Receive operations on ports can block in the kernel, thus providing a blocking synchronization mechanism.

Parallelism is realized through the use of multiple threads to implement a single application. Many different styles of communication and synchronization can be utilized by a collection of cooperating threads under PLATINUM. Communication between threads can use 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 into the shared space. Alternatively, a memory object can be mapped into multiple address spaces and thus be shared by all of the threads in those spaces. A shared memory object need not be mapped at the same virtual address range in every address space referencing it, nor do the access rights have to be uniform.

All PLATINUM kernel primitives are synchronous. They do not return until the requested operation is completed.

3.1 Rationale for the PLATINUM Model

The success or failure of NUMA architectures will rest on their ability to compete with supercomputers and multicomputers on the bases of performance and programmability in the execution of such computation-intensive programs as occur in scientific and artificial intelligence applications. PLATINUM is therefore aimed at user programs of this sort. As such, it provides a small set of very efficient communication and synchronization primitives with an emphasis on shared memory. Although the model is general-purpose, the current implementation of PLATINUM does not include support for protection and persistent storage required of an operating system for machines used by competing and perhaps hostile user programs.

We repeat our intention that PLATINUM be a simple platform for developing and experimenting with that part of the memory management system which deals specifically with NUMA. The PLATINUM project intends neither to design and implement a radically different general-purpose operating system nor to present users with a new conceptual model

of virtual memory. The goal is, rather, to concentrate our efforts on research issues directly related to the design, implementation, and evaluation of the memory management aspects of operating systems for NUMA multiprocessors. A secondary goal is to ensure that if our implementations of NUMA memory management prove successful, they can also be of widespread utility. One way of ensuring this is to anticipate the integration of the NUMA memory management subsystem of PLATINUM with an existing general-purpose operating system. Mach [22] is the logical target for this exercise. The model of virtual memory that PLATINUM presents to user programs is therefore derived from Mach. PLATINUM's virtual memory interface is a subset of the Mach virtual memory interface, and much of the code that implements the interface is derived from Mach sources.

4 Data Types

PLATINUM and all current applications are written in C++. From user programs all PLATINUM objects (threads, memory objects, address spaces, and ports) are referenced with unique 32-bit identifiers created by the kernel. Names can be compared, copied, placed in shared memory and passed in messages. The only protection on names is provided by C++ type checking.

```
typedef object_id_t     thread_t;
typedef object_id_t     port_t;
typedef object_id_t     address_space_t;
typedef object_id_t     memory_object_t;
```

In addition to object names, the following data types appear in the argument lists to kernel operations.

```
typedef short   priority_t;
typedef int     vm_offset_t;
typedef unsigned vm_size_t;
const vm_size_t vm_page_size;

enum            vm_prot_t {
                vm_prot_read    = 1,
                vm_prot_write   = 2,
                vm_prot_execute = 4}

const vm_prot_t vm_prot_default = vm_prot_read | vm_prot_write;
```

A **vm_offset_t** is an address in virtual memory and is represented as a signed offset with respect to an address space. The actual range of meaningful addresses is machine-dependent. The type **vm_size_t** is used to express the sizes of memory objects. The ratio of the size of a virtual memory page (**vm_page_size**) to the size of the machine page must be a non-negative power of two. All addresses and sizes are expressed in terms of bytes.

The access rights to a memory object within an address space are encoded by OR'ing together the appropriate set of protection bits as defined by vm_prot_t.

The codes returned by kernel operations indicate success or failure.

```
enum            return_t {
                failure = 0,
                success = 1,
                warning = 2
}
```

5 Threads

PLATINUM threads are bound to a single processor at any given time. This binding is under the complete control of the application. The initial binding is set at thread creation time, but the application can change it later. A change in binding results in the thread's migration to another processor. The kernel, however, never migrates a thread without a directive from the application.

Some multiprocessor operating systems such as Mach perform dynamic load-balancing among the processors through automatic thread migration. Automatic thread migration, however, makes it more difficult to study the effects of sharing on the memory management system. When a thread migrates the numbers of both cache misses and invalidations are likely to increase. This increase corresponds, respectively, to the replication of pages already at the previous location and to writes to pages still replicated there. For experimental purposes we want to be able to separate cache interference due to thread migration from cache interference due to sharing. Thus, PLATINUM does not provide automatic thread migration.

5.1 create_thread

```
return_t        create_thread(
                address_space_t address_space,
                vm_offset_t     program_counter,
                vm_offset_t     stack_pointer,
                priority_t      priority,
                unsigned char   *node,              // IN/OUT
                thread_t        *thread)            // OUT
```

This primitive creates a new thread and returns its name. The thread will execute in the address space specified by **address_space** and starts in the initial state specified by **program_counter**, **stack_pointer**, and **priority**. The new thread is created in the suspended state and must be explicitly started using the **resume_thread** primitive in order to execute. If **node** is specified, the thread is created on the specified processor. Otherwise, the kernel uses a static load-balancing heuristic to select the processor on which to create the thread. The new thread's priority can be no higher than that of the creating thread.

Threads are scheduled strictly by priority. On each processor the highest priority runnable threads are scheduled in a round-robin fashion. If a thread of higher priority becomes ready on a processor, the running thread is immediately preempted. This can occur both when a high priority thread is unblocked by another processor or device and when a thread's priority is increased by a thread running on another processor.

5.2 destroy_thread

```
return_t        destroy_thread(
                thread_t        thread)
```

The specified thread is destroyed.

5.3 migrate_thread

```
return_t        migrate_thread(
                thread_t        thread,
                unsigned char   *node)          // IN/OUT
```

This primitive binds the thread to a new location. If node is specified, the thread is moved to the specified processor. Otherwise, the kernel uses a static load-balancing heuristic to choose a destination processor. In the latter case it is possible that the thread will not move.

5.4 set_priority_thread

```
return_t        set_priority_thread(
                thread_t        thread,
                priority_t      priority)
```

A thread's priority can be set no higher than that of the thread changing its priority.

5.5 suspend_thread

```
return_t        suspend_thread(
                thread_t        thread)
```

The specified thread is placed in a suspended state. It cannot run in user state until it is resumed. Kernel primitives called by the thread are allowed to complete. Thus, a suspended thread that is blocked because it is executing a receive on a port retains its position in the queue and can become unblocked if it reaches the head of the queue and a message arrives. The thread remains suspended. Suspend_thread returns `failure` if the specified thread is already suspended.

5.6 resume_thread

```
return_t        resume_thread(
                thread_t        thread)
```

The thread is taken out of the suspended state. If the thread is not blocked it is made runnable. Otherwise, it continues to wait. **Resume-thread** returns **failure** if the specified thread is not in a suspended state.

Suspend and **resume** are intended to be used by user-level process management.

6 Messages and Ports

PLATINUM ports are simple, fast message queues. Any number of threads may send to or receive messages from a port. Network operating systems such as Mach, however, restrict the set of receiving threads to those within a single task (address space). This can be attributed to the difficulty of implementation when receiving threads are running on different machines in the network. Because they do not have this restriction, PLATINUM ports can be used as a blocking synchronization mechanism for data in memory objects shared by multiple address spaces.

A message is a variable-length array of zero or more bytes. The maximum size of a message is **vm_page_size**. The kernel does not interpret the message data. In particular, it performs no type checking. All PLATINUM object identifiers can be transmitted freely through messages or shared memory. Although the message-passing mechanism does not itself implement the automatic transmission of out-of-band data, an application can include the identifier for a memory object in a message and map it in the receiving address space.

In addition to ports allocated by user programs the kernel provides a set of "well-known" ports. These serve as the interface to the system's input/output services.

6.1 create_port

```
return_t        create_port(
                port_t          *port)                  // OUT
```

This primitive creates a new port and returns its name.

6.2 destroy_port

```
return_t        destroy_port(
                port_t          port)
```

The specified port is destroyed. All threads waiting on the port when it is destroyed will be unblocked with a status code of **failure**.

6.3 receive_message

```
return_t        receive_message(
                port_t          port,
                vm_offset_t     buffer,
                vm_size_t       *size)                  // IN/OUT
```

This primitive removes the message at the head of port's queue. If no message is available, the thread blocks. Blocked threads are served in first-come, first-served order. If the message at the head of the queue is larger than the buffer, the message remains on the queue and a **warning** status code is returned.

6.4 send_message

```
return_t        send_message(
                port_t          port,
                vm_offset_t     buffer,
                vm_size_t       size)
```

The sending of a message is non-blocking.

6.5 send_receive_message

```
return_t        send_receive_message(
                port_t          send_port,
                vm_offset_t     send_buffer,
                vm_size_t       send_size,
                port_t          receive_port,
                vm_offset_t     receive_buffer,
                vm_size_t       *receive_size)          // IN/OUT
```

This operation combines a **send** and a **receive**. It is intended for use in implementing remote procedure call and other synchronous message-passing protocols. If the message at the head of the queue is larger than the buffer, the message remains on the queue and a **warning** status code is returned.

7 Memory Objects and Address Spaces

PLATINUM has two types of memory object. Named memory objects are created explicitly by user programs. Anonymous memory objects are created by the kernel when a virtual address range is allocated but not mapped to a named memory object. Anonymous memory objects are zero-fill memory that is private to a single address space.

The process of mapping a virtual address range to a contiguous range of pages in a memory object creates a reference to that object. A memory object persists as long as there is at least one reference to it. A name counts as an additional reference, so a named memory object can survive even though it is not currently mapped into any address space. Destroying an address space or deallocating a virtual address range can remove a reference to a memory object. If an entire virtual address range mapping a memory object in an address space is deallocated, then a reference to the memory object is removed. If only a subrange of a virtual address range mapping a memory object is deallocated, then the range is either clipped at the appropriate end or split into two ranges. A split creates an additional reference to the memory object.

7.1 create_memory_object

```
return_t        create_memory_object(
                vm_size_t       size,
                memory_object_t *memory_object)         // OUT
```

This primitive creates a new memory object and returns its name. The newly created object is not in any address space. It is filled with zeros.

7.2 destroy_memory_object

```
return_t        destroy_memory_object(
                memory_object_t memory_object)
```

This primitive removes the binding between a name and the memory object to which it refers. The object can no longer be accessed through the name. The resources associated with the now anonymous memory object can be reclaimed when the last reference to it is removed.

A named object will be retained even if no mapping to it exists.

7.3 create_address_space

```
return_t        create_address_space(
                address_space_t *address_space)         // OUT
```

This creates a new address space and returns its name.

7.4 destroy_address_space

```
return_t        destroy_address_space(
                address_space_t address_space)
```

Destroying an address space destroys all threads running within it. This implicitly removes the mappings to its memory objects, thus potentially allowing them to be destroyed or reclaimed.

7.5 allocate_range_with_memory_object

```
return_t        allocate_range_with_memory_object(
                address_space_t address_space,
                vm_offset_t     *vadr,                  // IN/OUT
                vm_size_t       size,
                boolean         anywhere,
                memory_object_t memory_object
                vm_offset_t     memory_object_offset)
```

This binds a window of pages beginning at **memory_object_offset** within a memory object to a range of addresses in the specified address space. If **vadr** is provided, it will be truncated to the nearest virtual page boundary, as will **memory_object_offset**. **Size** will be rounded upward to the next virtual page boundary. If **anywhere** is true, the kernel will allocate the first available region of sufficient size. Otherwise, it will return **failure** if **vadr** is not the start of a sufficiently large region. Access to the range is set to **vm_prot_default**.

7.6 allocate_range

```
return_t        allocate_range(
                address_space_t address_space,
                vm_offset_t     *vadr,                  // IN/OUT
                vm_size_t       size,
                boolean         anywhere)
```

This allocates a region of virtual address space and creates an anonymous memory object which is mapped into that region. Since the memory object is unnamed there is no way that it can be mapped into any other address space. If **vadr** is provided, it is truncated to the nearest virtual page boundary. **Size** is rounded upward to the next virtual page boundary. If **anywhere** is true, the kernel allocates an available region of sufficient size. Otherwise, it returns **failure** if **vadr** is not the start of a sufficiently large region. Access to the range is set to **vm_prot_default**. The virtual memory region allocated is filled with zeros.

7.7 deallocate_range

```
return_t        deallocate_range(
                address_space_t address_space,
                vm_offset_t     vadr,
                vm_size_t       size)
```

This primitive deallocates the specified address range. Vadr is truncated to the nearest virtual page boundary. Size is rounded upward to the next multiple of the virtual page size. The deallocation of the range can remove references to one or more objects and potentially cause them to be destroyed. Deallocating an address range in the middle of a range to which an object is mapped splits the existing reference to the object into two.

7.8 protect_range

```
return_t        protect_range(
                address_space_t address_space,
                vm_offset_t     vadr,
                vm_size_t       size,
                vm_prot_t       prot)
```

Vadr is truncated to the nearest virtual page boundary. Size is rounded upward to the next multiple of the virtual page size. Access rights to the specified range of pages in the address space are set to the requested value.

8 Experiments with NUMA Memory Management.

As stated in section 2, the problem of providing a form of uniform shared memory model on a physically distributed shared memory machine can be attacked at several levels:

- Memory coherence can be implemented in the machine architecture level using hardware caching, as in UMA multiprocessors.
- It can be implemented transparently in the operating system kernel level as part of the memory management system.
- It can be implemented in user programs, either explicitly by the programmer or implicitly as part of the programming language model.
- The implementation can span two or more levels.

While PLATINUM primarily addresses the feasibility of a software implementation in the kernel, it also interacts with the layers above and below it. Experiments with PLATINUM cannot test the kernel in isolation; they must also shed light on these interactions.

The version of PLATINUM described in this paper is an experiment in providing uniformity by maintaining a coherent memory abstraction entirely and transparently in the operating system kernel on an existing NUMA architecture. The user program is not able to direct the execution of the coherency mechanism, nor is it able to affect the policies directing the mechanism. We intend to explore this approach thoroughly before examining other strategies for providing uniformity. This will include the tuning of the implementation of coherent memory as well as analyzing in detail the performance of the system running

user programs with widely varying patterns of shared memory usage. At one extreme, applications performing fine-grain modification on very large amounts of shared data will not benefit from the potential of migrating and replicating that data. At best we can hope to reduce the overhead incurred in recognizing this situation. At the other extreme, programs that perform relatively few modifications on shared data will place few demands on the coherency protocol. By characterizing the behavior of the memory manager on a variety of programs that lie between these two poles, we intend to define the domain for which this approach is effective.

PLATINUM is being instrumented to record the kind of detailed information we will need to analyze these experiments. Internally, the kernel uses simple locking protocols to synchronize access to its data structures. Fine-grain synchronization traces can be recorded using instrumented versions of the locking operations [18]. These can then be analyzed off-line using a suite of debugging and performance analysis tools that we have developed over the last two years at the University of Rochester [14]. The traces will be useful for tuning the kernel as well as recording the interleaving of kernel operations for the purpose of evaluating the performance of the memory management system.

Programmers and compiler writers concerned with performance try to tune their code for the architecture on which it is run. Although the implementation of PLATINUM coherent memory is entirely within the kernel, one effective mechanism for tuning user programs running on top of PLATINUM will be the careful placement of data within memory objects to reduce inter-processor interference. Performance can be adversely affected by co-locating in one page of coherent memory data items that have radically different properties with respect to sharing. Co-locating the private data of threads on distinct processors induces spurious sharing. Co-locating data items that are shared at different temporal granularities can lead to location choices inappropriate for each. For example, processors busy waiting on a lock will attempt to modify it using `test-and-set` at a much finer granularity than the data it protects. If the lock and the data are both on the same page, the contention for the lock will make it appear that there is a finer granularity of sharing for the data than is the case. This can prevent the data from being moved to the processor accessing it. These effects will have to be considered in the evaluation of PLATINUM.

Since programmers and compilers will attempt to be clever about the placement of data in memory, we will evaluate the effect of adding a mechanism for declaring the sharing properties of pages in coherent memory. A declaration will be used to parameterize the policy for migrating and replicating those pages. In the extreme, it will be used to disable data movement.

Implicit memory hierarchy management schemes are all based on the locality of memory references. When a program is in equilibrium, access patterns in the near future will be similar to those of the recent past. If a program executes in phases and access patterns change between phases, there will be transient decrease in the performance of the memory hierarchy at each change [28]. If phase changes are frequent and the mechanism to detect them reacts comparatively slowly, the system will never achieve equilibrium. Because coherent memory relies heavily on software, we expect that this will be one of the limitations of PLATINUM. A possible method of improving the response to transients without special architectural support is to provide a mechanism by which directives from the user program

can mark its phase changes for the kernel. For example, the act of releasing a lock usually marks the end of a phase of intense access to a block of data by one processor. By executing an appropriate directive the user program can inform the memory manager that the recent activity should be ignored when deciding whether to migrate or replicate the data. Directives to the memory manager can be inserted explicitly by the programmer or implicitly by compilers and/or run-time libraries. A similar strategy is advocated by McNiven and Davidson for providing information to the block replacement policy of a hardware cache [20]. Extensions to the PLATINUM interface in this direction will improve its ability to support languages such as Emerald [16], which includes a form of language-directed migration of shared data.

The viability of kernel-supported coherent memory will depend upon the architecture on which it is implemented. Although PLATINUM does not require architectural support beyond that available on a Butterfly Plus, it does take advantage of the facilities available to it. For example, the processor–memory switch in the Butterfly Plus provides a block transfer primitive that moves data between memories at an incremental cost comparable to reading a local memory. Given an approximately fifteen-to-one ratio between the costs of remote and local access, replication and migration are usually desirable even if only a small fraction of the data moved is actually accessed at the destination. PLATINUM also makes extensive use of the flexibility of the Motorola MC68851 memory management unit.

We anticipate porting PLATINUM to a very different NUMA machine. The IBM ACE Multiprocessor Workstation has a global memory whose access cost is between that of accessing local memory and that of accessing another processor's memory remotely, a very different memory management unit, and no special block transfer mechanism. On this architecture the high relative cost of data movement will make it less attractive than on a Butterfly Plus. On the other hand, placing data in the common global memory is a useful alternative. These properties will affect the policies used within PLATINUM. The architectural decisions made for each machine can then be evaluated with respect to kernel-supported memory coherency by comparing the PLATINUM implementations.

Comparative architectural studies will also help predict the value in practice of other forms of architectural support. For example, analytic studies [5, 24] have assumed the existence of reference counters that are sensitive to the relative locality of the processor generating each reference. Such counters are not part of any existing machine. Before actually designing shared memory management and caching hardware that includes coherent reference counts it is worth while to attempt to understand the performance of systems without them.

9 Status and Conclusions.

At this time PLATINUM is running on BBN Butterfly Plus Multiprocessors. Since all kernel data structures that are not a part of the coherent memory system are stored in coherent memory, memory management had to be functional before we could finish the kernel. Consequently, it is the most thoroughly tested part of the kernel. Furthermore, building the remainder of the kernel on the coherent memory made the kernel's design and

development an easier task than the construction of Osiris, a similar experimental NUMA kernel implemented as a pedagogical exercise.

PLATINUM is a small kernel. We are now in the process of designing and implementing a more complete operating system interface that will provide basic services such as file and terminal I/O. These services will be provided by a set of servers communicating with applications through message-passing and shared memory.

We have begun work on the development of applications to run on PLATINUM. A simulator for recurrent backpropagation networks [25] is the first real application to be ported.

Acknowledgements.

We thank Lawrence Crowl and Tom LeBlanc for their helpful comments and suggestions. Special thanks go to Niki Hansen for her editorial assistance.

References

[1] A. Agarwal and A. Gupta, "Memory-Reference Characteristics of Multiprocessor Applications under Mach", *Performance Evaluation Review*, May 1988.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE, June 1988, pp. 280–289.

[3] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, volume 4, number 4, November 1986, pp. 273-298.

[4] J. Archibald, "The Cache Coherence Problem in Shared-Memory Multiprocessors", February 1987.

[5] D. Black, A. Gupta, and W.D. Weber, "Competitive Management of Distributed Shared Memory, *Spring Compcon*, 1989.

[6] W. Bolosky, personal communication.

[7] "Butterfly Parallel Processor Overview", BBN, Cambridge, Massachusetts, June, 1985.

[8] "The Uniform System Approach To Programming the Butterfly Parallel Processor", BBN, Cambridge, Massachusetts, October 1985.

[9] "Inside the Butterfly Plus", BBN, Cambridge, Massachusetts, October 1987.

[10] M. Censier and P. Feautier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE TC*, C-27(12):1112-1118, December 1978.

[11] D. Cheriton, G. Slavenburg, and P. Boyle, "Software-Controlled Caches in the VMP Multiprocessor", *Proceedings of the 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, June 1986, pp. 366-374.

[12] D. Cheriton, A. Gupta, P. Boyle, and H. Goosen, "The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation", *Proceedings of the 15th Annual Symposium on Computer Architecture*, June 1988, pp. 410-421.

[13] A. Cox and R. Fowler, "A Memory Management System for NUMA Multiprocessors", Technical Report 263, Department of Computer Science, University of Rochester, in preparation.

[14] 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". *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distribute Debugging*, Madison, Wisconsin, May 1988.

[15] E.F. Gehringer, D.P. Sieworek, and Z. Segall, "Parallel Processing: The CM* Experience", Digital Press, 1987.

[16] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems.*

[17] T. J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study", Butterfly Project Report 3, Department of Computer Science, University of Rochester, January 1986. (A shorter version appears in *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986, pp. 463–466.)

[18] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, C-36(4):471-482, April 1987.

[19] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors", Yale/DCS/RR-492, September 1986.

[20] G. D. McNiven and E.S. Davidson. "Analysis of Memory Referencing Behavior for Design of Local Memories", *Proceedings of the 15th Annual Symposium on Computer Architecture*, June 1988, pp. 56–63.

[21] G.F. Pfister, W.C. Brantley, D.A. George, S.L Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, A. Norton, J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp. 764–771

[22] 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", *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987, pp. 31–39.

[23] Z. Segall and L. Rudolph, "Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor", *Proceedings of the 11th International Symposium on Computer Architecture*, June 1984, pp. 340–347.

[24] C. Sheurich and M. Dubois, "Dynamic Page Migration in Multiprocessors with Distributed Global Memory", in *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1988, pp.162–169.

[25] P. Simard, M. Ottaway, and D. Ballard, "Analysis of Recurrent Backpropagation", *Connectionist Summer School Proceedings 1988.*

[26] R. L. Sites and A. Agarwal, "Multiprocessor Cache Analysis Using ATUM", *Proceedings of the 15th International Symposium on Computer Architecture*, May 1988, pp. 186–195.

[27] L. Snyder, "Type Architectures, Shared Memory, and the Corollary of Modest Potential", *Annual Review of Computer Science*, Volume 1, 1986, pp. 298–317.

[28] W.D. Strecker, "Transient Behavior of Cache Memories", *ACM Transactions on Computer Systems*, 4(1), November 1983, pp.281-293.

[29] W.A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, New York, McGraw-Hill, 1981.

[30] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proceedings of the Eleventh Symposium on Operating Systems Principles*, November 1987, pp. 63–76.