Operating system support for parallel programming on RP3

by R. M. Bryant
H.-Y. Chang
B. S. Rosenburg

RP3, the Research Parallel Processing Prototype, was a research vehicle for exploring the hardware and software aspects of highly parallel computation. RP3 was a shared-memory machine that was designed to be scalable to 512 processors; a 64-processor machine was in operation from October 1988 through March 1991. A parallel-programming environment based on the Mach operating system was developed, and a variety of programming models were tested on the machine. To help user programs realize the full potential of parallelism on RP3, the RP3 operating system was extended to support such RP3 architectural features as noncoherent caches, local and interleaved storage, and a hardware performance monitor. The system included explicit job-scheduling and processor-allocation facilities, facilities for exploiting the RP3 memory hierarchy, and performance-data collection and logging facilities. This paper describes these components of the RP3 operating system, provides the rationale for the design decisions that were made, and discusses the implementation of these operating system facilities.

Introduction

RP3, the Research Parallel Processing Prototype, was a research vehicle for exploring the hardware and software aspects of highly parallel computation. RP3 was a shared-memory machine that was designed to be scalable to 512-way multiprocessing; a 64-way machine was built and was in operation from October 1988 through March 1991.

To allow efficient shared-memory parallel programming, the RP3 architecture included several features designed to minimize memory conflicts and spread memory references as uniformly as possible across the machine. The intent of these features was to eliminate (as much as possible) bottlenecks in the hardware that would keep applications from achieving acceptable parallel performance. For similar reasons, although each processor on the machine had a 32-kilobyte (KB) cache, there was no hardware support for keeping the contents of caches on different processors consistent. Instead, it was intended that application code would issue explicit cache-invalidation

operations to ensure consistent access to shared data via the cache. The architecture also allowed *noncached* access to data for which explicit cache management was not feasible. In summary, the RP3 architecture was designed to give application code direct control over the hardware in order to let applications achieve maximum parallel performance. This design philosophy was carried forward in the design of the RP3 operating system.

Early in the project, it was decided that RP3 should run a version of the Berkeley Software Distribution (BSD) UNIX® operating system [1]. The rationale for this decision was that

- BSD UNIX systems were in common use at many universities. (One of the goals of the RP3 project was to encourage collaboration with university researchers in the field of parallel processing.)
- The BSD system was thought to be relatively easy to port, and source code for the system was readily available.
- The UNIX system has features (e.g., pipes, fork) useful for parallel processing.
- Many people are familiar with the UNIX programming interface; a somewhat smaller group is familiar with UNIX system internals.

Given this decision, the RP3 operating system designers were faced with the challenge of extending the UNIX programming interface to support the kind of experimentation and usage that was envisioned for RP3.

In the remainder of this paper, we describe the RP3 architecture, especially those features of the architecture that required special operating system support. We then discuss the major programming models the RP3 operating system was expected to support, and we justify our choice of the Mach operating system [2] from Carnegie Mellon University (an operating system compatible with the BSD UNIX system) as the UNIX system to be ported to RP3. Finally, we discuss the extensions to Mach that we made for RP3: parallel-program-scheduling support, support for the RP3 memory model, and performance-measurement support.

This paper focuses on the definition and implementation of the operating system extensions we made. An evaluation of these extensions and a discussion of our experience in implementing the RP3 operating system can be found in [3].

RP3 hardware overview

Figure 1 illustrates the RP3 architecture. An RP3 machine could consist of up to 512 *processor-memory elements* (PMEs). The prototype hardware that was actually built, called *RP3x*, consisted of 64 PMEs. Each PME included the following components:

CPU	Central processing unit, a 32-bit RISC processor. The same processor was used in the original IBM RT Personal Computer.®
FPU	Floating-point unit, similar to the floating-point unit found in second-generation IBM RT Personal Computers. It used the Motorola MC68881 floating-point chip, which implements the IEEE floating-point standard.
I/O	I/O interface, which provided a connection to an IBM PC/AT® system that served as an I/O and Support Processor, or ISP. Each ISP was connected to eight PMEs; all ISPs were also connected to an IBM System/370™ mainframe.
MMU	The memory-management unit. The MMU provided a typical segment and page-table address-translation mechanism and included a 64-entry, two-way-set-associative translation lookaside buffer (TLB).
Cache	A 32KB, two-way-set-associative, real-address cache. To allow cache lookup to proceed simultaneously with virtual-address translation, the RP3 page size was made equal to the cache set size of 16 KB.
MC	Memory controller. The memory controller examined each memory request to determine whether it was for this PME (in which case it was passed to the memory module) or a remote PME (in which case it was passed to the switch). The first nine bits of the address specified the target PME.
Memory	An 8-megabyte memory module.
PMC	Performance-measurement chip. This device included registers that counted such things as instruction completions, cache hits and misses, local and remote memory references, and TLB misses. It could also periodically sample the switch response time. Use of the PMC under Mach/RP3 is further described below in the section on operating system support for performance measurement on the RP3.

All the PMEs of an RP3 machine were connected by a multistage *interconnection network* or *switch*. The switch, which was constructed using water-cooled bipolar technology, had 64-bit data paths and a bandwidth of roughly 14 megabytes per second per PME.

All memory on RP3 was local to individual PMEs but was accessible from any PME in the machine. However, a performance penalty was incurred by a PME when accessing remote memory. RP3x had an access-time ratio

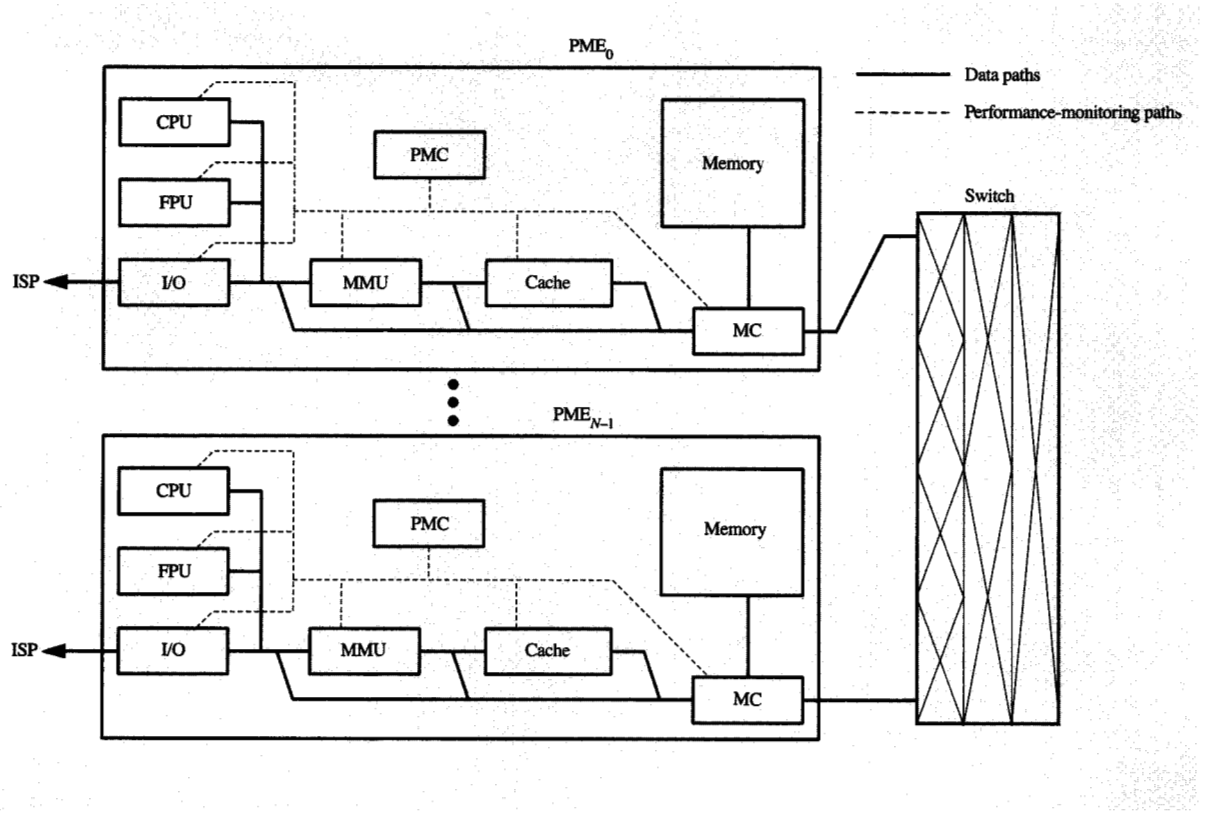

Figure 1

The RP3 architecture.

of 1:12:20 between cache, local, and remote memory, assuming no switch or memory contention. The fact that not all memory in the system had the same access time put RP3 in the class of *nonuniform-memory-access* (NUMA) machines. Other machines in this class include the BBN Monarch [4] and the Illinois Cedar [5] parallel processors. Support of the RP3 NUMA architecture required operating system extensions that are discussed below in the section on operating system support for the RP3 memory architecture.

To spread memory references evenly across memory modules (and thus decrease the chance of encountering memory bottlenecks), the RP3 memory-management unit supported *interleaved* pages. That is, addresses for interleaved pages underwent an additional transformation after virtual-to-real address translation. The interleaving transformation exchanged bits in the low- and high-order portions of the real address [see **Figure 2(a)**]. Since the high-order bits of the address specified the PME number, the effect of the interleaving transformation was to spread interleaved pages across memory modules in the system, with adjacent double-words being stored in different memory modules. The number of bits interchanged (hence the base-2 logarithm of the number of modules used to store the interleaved page) was specified by the interleave amount in the page table. **Figure 2(b)** shows how the interleaving transformation could be used to spread the pages of a virtual address space across multiple PMEs. For example, the first page of the illustrated virtual address space (at the top) is mapped with an interleave amount of two, so the page actually occupies a fourth of a physical page in each of four adjacent PME memory modules. The real address (before interleaving) to which the virtual page is mapped determines which PMEs and which regions within those PMEs the virtual page will occupy. (The diagram is somewhat simplified, in that interleaved pages do not really occupy contiguous regions of physical storage. A virtual page mapped with an interleave amount of two, for example, will actually occupy every fourth double-word of a physical page in each of four memory modules, rather than a contiguous fourth of each page.)

Normally, all variables used by more than one PME were stored in interleaved memory. For this reason, interleaved memory was also referred to as *global*

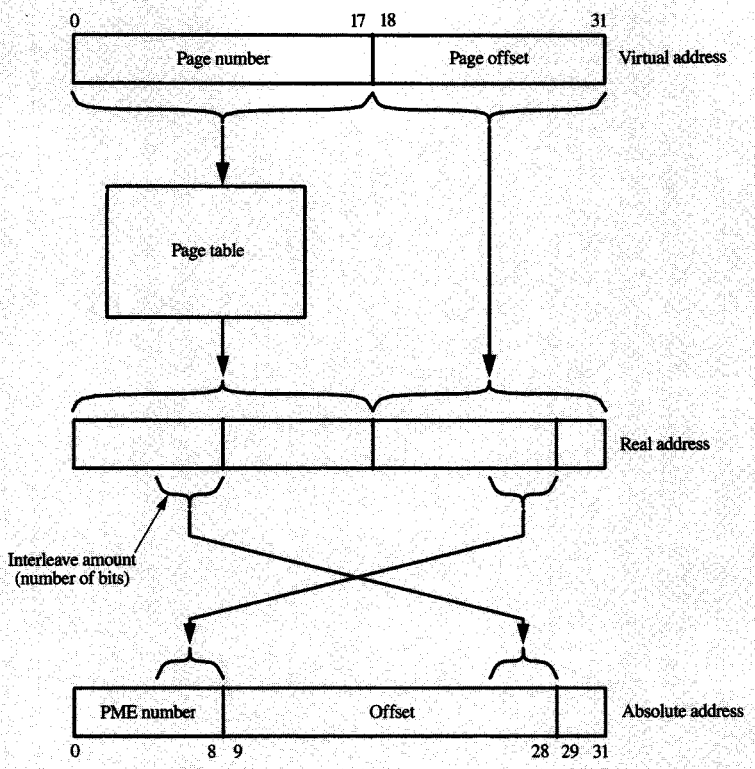

Figure 2

The RP3 interleaving transformation: (a) Virtual address to absolute address translation. (b) Example of interleaving.

memory. Noninterleaved memory was referred to as *local* memory. (All memory, of course, was packaged with one PME or another.)

If enabled in the hardware, a one-to-one hashing transformation was applied before the interleaving transformation. The hashing transformation randomized sequential memory references as an additional technique to minimize memory conflicts. The hashing transformation was transparent to application and operating system software, so it is not further discussed here (see [3] for experience with the hashing transformation on RP3).

The RP3 hardware did not provide any mechanism for keeping caches coherent among PMEs; cache coherency had to be maintained in software. The page tables included cacheability information specifying which pages were to be accessed through the cache mechanism (cacheable pages) and which pages were not (noncacheable pages). Since there was no page table associated with real-mode memory access, all real-mode memory accesses on RP3 were noncacheable references. The cache was visible to application code, in the sense that user-mode instructions to invalidate all or parts of the cache were provided. Cacheable memory could be further identified as *marked data*. A single cache operation could be used to invalidate all data in the cache that had been loaded from virtual memory identified as marked data.

RP3 provided the *fetch&add* [6] operation (as well as *fetch&or*, *fetch&and*, etc.) as the basic synchronization primitive. *Fetch&add(location,value)* is an atomic operation that returns the contents of "location" and then increments the contents of the location by "value."

Further details of the design of the RP3 PME and system organization can be found in [7] and [8]. Details of how the prototype RP3x differed from this published design are given in [3].

Parallel-programming models

The RP3 operating system was expected to support a variety of parallel-programming models. Such models define the nature of the individual components of a parallel program and the manner in which the components access shared memory. Here we discuss three models of parallel computation: the process model, the task/thread model, and the thread/work-queue model.

- *Process model*

In many multiprocessor versions of the UNIX operating system, parallel programs are composed of multiple UNIX processes, typically one per real processor on the system, with each process having access to a region of shared storage where global variables are placed. Multiple processes are created using the UNIX *fork* system call; a child process is created for each processor in the system. Since the processes are independently schedulable entities, they run concurrently on separate processors, provided that enough processors are available.

Because of the expense of the *fork* system call, it is more efficient to create the child processes once, at the start of program execution, rather than to create and destroy them as the program runs. User-level synchronization primitives are used to coordinate the child processes.

In this programming model, the child address spaces are independent and private except for a common global segment that is shared among all the child processes. Code that is identical in each child may be shared as well.

There are several advantages to this approach to parallel processing. First, system support for this model is easy to implement. Beyond basic support for sequential processes, all that is necessary are facilities for creating and accessing a shared storage segment and for using the basic synchronization primitives of the underlying hardware (*test&set*, *compare&swap*, or *fetch&add*, for example). Second, this programming model is supported by a number of different operating systems, so applications can be readily ported from one environment to another. Finally, this approach to parallel programming requires no special compiler or language run-time support. The entities that comprise a parallel program are simple sequential processes with private address spaces.

The EPEX [9] parallel-programming environment uses the process model of parallel computation. This programming environment was developed to let programmers experiment with RP3 applications long before the RP3x prototype was available. The EPEX environment was first implemented under VM/370 on multiprocessor System/370 mainframes. It was later ported to the IBM RT PC® workstation and then to RP3 itself. A significant collection of RP3 applications were developed under the EPEX environment.

The disadvantages of the UNIX process model of parallel computation are discussed in the following subsections.

Private address spaces Except for the single shared segment, the child process address spaces are private. Variables stored outside the shared region are not accessible to other processes in the program. Since the program stack is normally in the private area, automatic (local) variables within the program cannot be shared between processes. In addition, the existence of private variables can make a process special in the sense that it is the only process that can complete a particular portion of a job.[1] It may therefore be impossible for the operating system to remove a processor from the program without

[1] In this paper, we use the terms "job," "application," and "program" interchangeably.

causing the parallel program to become deadlocked or otherwise fail.

Local resource ownership Because in the UNIX system the process is the entity that owns resources, resources cannot be shared among the different parts of a parallel program. For example, a file opened by one child process is not accessible to other child processes, unless they open the file as well. This particular problem is avoided if the file is opened in the parent before the child processes are created, but such *a priori* acquisition of resources is not always possible. The result of this limitation is that often one must designate a particular process to handle all input and output for the entire parallel application, and this designated process can easily become an unacceptable serial bottleneck in a highly parallel program.

Heavyweight processes In standard UNIX systems, the only way to make use of a processor is to create a process and its associated address space. This restriction can make changing the number of processors allocated to a particular program prohibitively expensive. This problem is not severe if only one job is executed at a time, but for multiple-job parallel processing, processor reallocation is occasionally required. Furthermore, in the typical UNIX model, it is not clear how to organize the code to take advantage of a newly available processor or to recover when a processor is removed (because of the private-data problem mentioned above). In addition, start-up overhead can be significant when many processors are involved and large numbers of address spaces must be created.

These problems can be solved by using the task and thread constructs of the Mach operating system [2].

- *Task/thread model*

In Mach, the UNIX process construct is split into *task* and *thread* primitives. The *task* is the entity that owns resources. It has an address space and a UNIX process identifier, and it can own communication port rights—software capabilities that allow tasks to send data to and receive data from other tasks. The *thread* is the entity that executes code; it can be thought of as a lightweight process. A thread executes in the context of a single task throughout its life, but many threads may belong to the same task. In a multiprocessor, the threads can execute simultaneously on separate processors. The resources of a task are available to all the threads that belong to it. A UNIX process is emulated in Mach by a task with a single thread.

Thus, in Mach a parallel application can be structured as a number of threads running within a single task. Each thread needs an individual work area within the shared address space, but all storage is accessible to all the threads. Addresses generated by one thread are equally valid for all threads, so work can easily be transferred from one thread to another. The threads also share resources such as communication port rights and open files, so no single thread becomes a serial bottleneck. Finally, since thread creation and destruction do not involve the address space or other resources, these operations are relatively inexpensive. An application can afford to vary its processor usage during the course of a computation.

The task/thread model does not by itself let the operating system vary the number of processors dedicated to a parallel application, but programs using the model can be structured in such a way that they can readily adapt to such changes.

- *Thread/work-queue model*

The thread/work-queue model of parallel computation is a refinement of the task/thread model that lets application programs adapt to changes in the number of processors available to the applications. The model has the following characteristics:

- Each parallel application (or job) runs in a single address space.
- Multiple threads may be created in the address space.
- A work queue is defined by the application.
- Each thread selects work to do by removing an entry from the work queue, executing that request to completion, and returning to the queue for the next work item.
- Execution of a work item can result in the addition of new items to the work queue.
- When new processors are allocated to the job, new threads are created to use them. These threads begin execution by selecting a work item from the work queue.
- When a processor is removed from a job, the thread running on that processor is suspended, and its execution state is saved in the work queue. The interrupted work item is completed by the next thread that examines the work queue.

This approach allows the user to specify how processors are used and to assign work to those processors in an application-dependent way. For example, in a Parallel FORTRAN program, a work item may be the execution of a single DO-loop iteration, and selection of the next work item may be as simple as taking the next loop index value from a global variable. In this case, the work queue implementation is trivial; more complex implementations are required if the work descriptor is more complicated than the value of an iteration variable.

If multiple jobs are executed at the same time, a "global processor allocation" component of the operating system is responsible for allocating processors among the jobs.

While the thread/work-queue model allows processors to be added to or removed from an application, these operations are still too expensive to be done every time the work queue is inspected. Global processor allocation decisions are therefore made on a medium-term basis (minutes, rather than seconds); thus, once a processor is allocated to a job it remains allocated to that job for a reasonable period of time.

On RP3, programs written using PTRAN [10–12] and the C-Threads package [13] used the thread/work-queue model of parallel computation.

Operating system support for the RP3 architecture

We adopted Mach as the base operating system for RP3 for a variety of reasons:

- Mach was compatible with BSD UNIX operating systems. As discussed in the Introduction, this compatibility was a requirement for the RP3 operating system.
- Mach was already available on the IBM RT PC workstation. (RP3 used the same processor as the RT PC.)
- Mach could run on shared-memory multiprocessors. Other BSD UNIX systems available to us were not multiprocessor-capable.
- Mach supported the UNIX process model of parallel computation. Such support was necessary because of the existing collection of EPEX applications.
- Mach supported the task/thread and thread/work-queue models of parallel computation. New RP3 applications were programmed using these models.

Choosing Mach as the base RP3 operating system allowed us to concentrate on specific operating system issues relevant to the RP3 architecture: multiprocessor scheduling, memory management, and performance measurement.

- *Multiprocessor scheduling*

One of the problems with the Mach scheduler (and UNIX schedulers in general) is that the operating system is not aware that the collection of threads[2] comprising a parallel program is an entity that should be scheduled as a unit. Instead, the operating system scheduler regards the threads in the parallel program as independent entities. Some of the problems that may arise are the following.

Inefficient work partitioning Under the EPEX [9] system, DO-loops are partitioned across processors by having each processor execute the DO-loop body for a subrange of the DO-loop index values. Then, at the end of its portion of the loop, each processor waits until all the other processors have completed their portions. The size of the subrange is calculated at run time by dividing the DO-loop range as equally as possible among the available processors. If the operating system changes the number of processors available to the program (for example, by suspending execution of one of the threads in the parallel program), the partitioning of the DO-loop is not optimal. To solve this problem, one can create an interface that allows the operating system to inform the application when one of its threads has been suspended [14]. Alternatively, the operating system can guarantee to the application that if any of the threads in the application are running on a processor, all of the threads are running. We followed the latter approach in Mach/RP3.

Excessive spin-waiting In order to protect the integrity of shared data, parallel programs must use a synchronization protocol. A locking protocol is an example of a synchronization protocol; *barriers* and *serial sections* [9] are other primitives used. When a thread attempts to acquire a lock that is not currently available, the thread must wait until the lock is released.

Waiting can be implemented as either *spin-waiting* or *suspend-waiting* (or some combination of these two approaches [15]). With spin-waiting, the waiting thread repeatedly tries to acquire the lock. With suspend-waiting, the waiting thread is suspended until the lock becomes available. While suspend-waiting is more efficient (the waiting thread is not using a processor), the cost of suspending and then resuming the thread may be higher than the cost of spin-waiting, particularly if the lock is held for only short periods of time. Many parallel programs use spin-waiting, because the critical sections protected by locks are very short.

This strategy works when there are enough processors to satisfy the demands of all active parallel programs. However, when multiple parallel programs share the machine, there is the possibility that at least one thread in a parallel program will be suspended while other threads of the program are running. The performance of the parallel program can deteriorate because of excessive spin-waiting by threads waiting for the suspended thread. Similarly, programming models such as EPEX [9] use a spin-waiting barrier at the end of each parallel loop. If a thread cannot reach the barrier because it has been preempted, all the rest of the threads in the job will spin-wait at the end of the loop, waiting for the last thread to complete the loop.

Poor job response time In a uniprocessor system, the shortest-job-first policy has the best job response time among the class of conservative scheduling algorithms. By

[2] This discussion applies to both the Mach task/thread model of parallel computation and to the UNIX process model. For the latter model, the terms *thread* and *process* are synonymous. A parallel computation consists of a number of threads; whether those threads have separate address spaces or share a single address space is irrelevant to the scheduling issue.

```
Family-specification routines:
   family_create(&family)
   family_remove()

Processor-allocation routines:
   processor_allocate(family, min_num_processors,
      max_num_processors, block)
   processor_deallocate(family, num_processors)

Thread-binding routines:
   thread_bind(thread)
   thread_addbind(bound_thread, thread)
   thread_unbind(thread)
```

Figure 3

The RP3 family-scheduling operating system interface.

analogy, in a multiprocessor system a policy that gives priority to jobs with lower total processing requirements should give the best job response time. In practice the computation times of jobs are not known in advance, and a round-robin policy is often used. As Leutenegger and Vernon [16] point out, a thread-based round-robin scheduling policy may perform much worse than a job-based policy, because round-robin among all threads will give preference to parallel programs that have more threads. However, such programs most likely require longer total computation times, violating the shortest-job-first scheduling principle.

RP3 family-scheduling interface

To solve these problems, the operating system had to provide a version of coscheduling [17] or gang scheduling [18]. We use the term *family scheduling* to describe the facility that was implemented in Mach/RP3. The routines of the RP3 family-scheduling interface are listed in **Figure 3**. They fall into three categories: family-specification routines, processor-allocation routines, and thread-binding routines.

In Mach/RP3, a *thread family* was a set of cooperating threads working toward a single goal. All of the threads in a particular parallel application would normally be part of the same thread family. These threads exchanged messages, synchronized computational steps, and shared part or all of their address spaces. The thread family was the largest schedulable unit in Mach/RP3. That is, all of the threads in a thread family were scheduled to run at the same time, and if any of the threads in the thread family were suspended by the operating system, all of the threads were suspended. The thread that created a family (via the *family_create* system call) was originally the sole member of the family. The family was then built through inheritance: Threads created by a member of a family (by the standard Mach mechanism) were automatically included in the family. A thread could remove itself from its family via the *family_remove* call. When a family was created, a Mach port (the *family port*) that represented the family was returned via the *family* parameter of the *family_create* call. Rights to this port could be passed to other threads. Any thread that had rights to a family port could request that processors be allocated to or deallocated from the thread family.

The *processor_allocate* system call allowed the user to request that a number of processors be allocated to a thread family. The *block* parameter of the *processor_allocate* system call indicated whether the requestor wished to be blocked if the processor allocation request could not immediately be satisfied, or to be given an error return code that indicated that not enough free processors were available to satisfy the request. If the requestor was blocked, the requestor's thread family was suspended as a whole until the required resources became available. The programmer could choose the appropriate interface, depending on whether the application had an absolute or flexible need for processing power. Processors could be deallocated from a thread family by means of the *processor_deallocate* system call.

Once processors had been allocated to a thread family, threads in the family were allowed to run only on those processors. Nonfamily threads were barred from the allocated processors. The threads of the family time-shared the family processors in the normal UNIX sense. On RP3 it was sometimes useful to "bind" threads to individual processors. An application could dedicate processing power to important threads by binding them to processors. Furthermore, a thread could use local storage if it knew it would not be moved from one processor to another. For a discussion of local memory allocation, see the section on operating system support for the RP3 memory architecture. A thread could be bound to one of its family's allocated processors with the *thread_bind* system call. Typically, a thread would "bind" itself. In this case, when the *thread_bind* request returned (successfully), the thread would be running on the bound processor. Note that a thread did not have to be bound to a processor in order to execute. An unbound thread in a family could execute on any available processor. Mach/RP3 also provided a *thread_addbind* call that let a thread be bound to the same processor as a previously bound thread. It was thus possible to bind several threads to one processor in order to share the processor and its local memory. A bound thread could be unbound using the *thread_unbind*

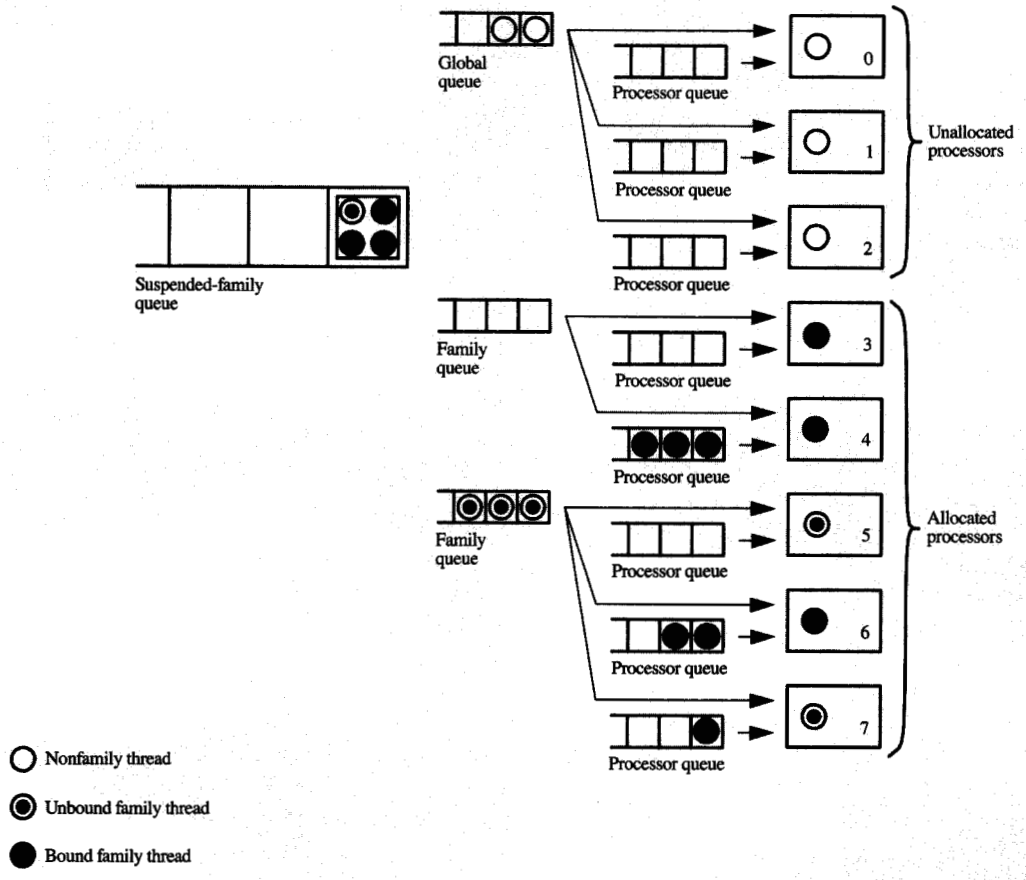

Figure 4

The RP3 family-scheduling mechanism.

system call. We decided not to use processor identifiers in the family-scheduling interface. Instead, the thread identifier of a bound thread served as a handle for a processor. If a user wished to allocate storage that was local to a processor, the user had to supply the thread identifier of a thread bound to that processor.

A notable feature of the RP3 family-scheduling interface was its flexibility in allowing threads to issue requests on behalf of other threads. For example, one thread could bind another thread, and a thread could allocate processors for a family to which it did not itself belong. This flexibility allowed a server thread to manage processor allocation and binding for a collection of families.

Scheduling mechanisms

The RP3 family-scheduling implementation required three categories of scheduling queues. The first consisted of one queue for each processor (the *processor queues*), containing the threads bound to that processor. The second category consisted of one queue for each family (the *family queues*), containing the unbound threads in that family, and a *global queue*, containing threads not belonging to any family. The third category consisted of a single queue of suspended families (the *suspended-family queue*). The processor and family queues contained only threads that were ready to run but were not currently executing. Threads that were currently executing were assigned to a processor.

Figure 4 illustrates the various scheduling queues. The diagram shows eight processors, the first three of which are unallocated. Each unallocated processor is executing a nonfamily thread; two other nonfamily threads are ready to run and are therefore in the global queue. Processors 3 and 4 are allocated to one family, while processors 5, 6, and 7

are allocated to another. The second family includes five unbound threads and four bound threads. Processor 5, to which no threads have been bound, is executing one of the unbound family threads. Three threads have been bound to processor 6, and one of them is executing. One thread has been bound to processor 7, but an unbound family thread is currently running on that processor. The other three unbound family threads are in the family queue. A third family, consisting of one unbound thread and three bound threads, has been suspended and is in the suspended-family queue.

In the first version of the family-scheduling mechanism, when a processor reached a rescheduling point (that is, when the currently executing thread was blocked or terminated), the processor would take the first bound thread from the processor queue. Only when the processor queue was empty would the processor take an unbound thread from its family queue. This priority scheme led to a deadlock situation in which an unbound thread could not release the locks that would let the bound threads proceed. (The bound threads were spin-waiting for locks that only the unbound thread could release.) This problem prompted a redesign that decoupled priority from thread binding. As a result, high-priority unbound threads could execute on family-owned processors even if lower-priority threads had been bound to those processors. Mach (and Mach/RP3) retained the UNIX dynamic-priority scheme that adjusts the dispatching priority of all threads to give long-waiting threads a chance to run.

This scheduling scheme did not apply to the "master processor," which—under the version of Mach we used on RP3—was reserved for system call service and therefore was not allocatable.

The suspended-family queue was a simple first-in-first-out queue. However, there were privileged system calls that provided an interface to a medium-term policy server that controlled family-level resource allocation. These calls allowed the policy server to suspend or resume an entire family. When a family was resumed, the original PMEs had to be reclaimed and the original thread bindings had to be restored, since the bound threads might have allocated local memory. (The potential impact of this restriction on machine utilization was never observed, because in practice RP3x usually ran a single job at a time.)

One problem that arose in the implementation of thread binding resulted from the fact that one thread could bind another. The semantics of the *thread_bind* call guaranteed that once the call returned, the target thread would be running on its bound processor. It was sometimes necessary to forcibly preempt the target thread from the processor on which it was currently running. The implementation of the *thread_bind* call used an interprocessor interrupt to accomplish this preemption.

Scheduling policies

The family-scheduling facility was a general facility that could meet the scheduling requirements of a broad range of system and user environments. For example, the family scheduler allowed multiple parallel applications to execute simultaneously in a production environment where system resource utilization might be very important. On the other hand, RP3x was most often used as a research machine for the measurement and evaluation of parallel algorithms. To obtain accurate measurements, interference among parallel applications had to be kept to a minimum. An application could use the facilities of the family scheduler to ensure that no other application would run at the same time. In practice, applications were supplied to RP3x via a batch server, and the batch server allowed a job to specify either shared or exclusive mode. In exclusive mode, a job had the machine to itself, while multiple jobs could coexist in shared mode, relying on the family-scheduling facility to equitably allocate resources.

The family-scheduling mechanism depended on parallel applications to properly specify their own processor-resource requirements. This dependence caused the waste of processing power for two reasons.

Wasted processing power inside an application While the family-scheduling interface allowed dynamic allocation and deallocation of processors, applications were not required to release unused processors, which resulted in wasted processor power within the application.

Wasted processing power due to processor allocation

Waste of processing resources could also occur when processors were available but none of the waiting families could run because there were not enough free processors to accommodate the total requirements of any waiting family.

The family-scheduling approach described in [19] avoids this problem by having the system control the sizes of individual families. Families are forced to reduce their processor requirements until they can coexist. The major shortcoming of this scheme is its failure to recognize that a job may not be able to utilize different numbers of processors equally efficiently. A better scheme is to choose family sizes through negotiation, in order to maximize the total efficient utilization of resources [20].

Another approach is to schedule the threads of a waiting family on all free processors in a round-robin manner. Because the guarantee that all members of a family will be run simultaneously is not maintained, performance may be poor, but at least the leftover processors are not completely wasted.

For RP3, we decided it was more important to give families all the resources they required than to optimize processor utilization. The system-call interface allowed the

application to adjust its processor-resource request in relation to the number of free processors. In addition, we felt that since there were a large number of processors on RP3, we did not need to be concerned with maximizing utilization of individual processors. Finally, we felt that this approach was most consistent with our philosophy of allowing user code to control the machine resources directly, thus allowing maximum experimental flexibility without operating system interference.

Comparison to the CMU processor-allocation scheme
Subsequent to the development of the family-scheduling interface in Mach/RP3, the Mach group at Carnegie Mellon University extended the system to support processor allocation [21]. In this section we compare the latter approach (the "CMU approach") to the Mach/RP3 family scheduler.

The CMU approach is based on *processor sets* rather than on thread families. Processors and threads can be assigned to or removed from processor sets. The threads assigned to a processor set are constrained to run only on the processors assigned to the set. If all processors are removed from a processor set, the threads assigned to that set are suspended. Processor allocation is accomplished by moving processors among processor sets.

The CMU approach clearly separates processor-allocation policy from mechanism. The processor-set mechanism is implemented in the operating system. Processor-allocation policy is implemented in a user-level server that moves processors and threads among processor sets. The policy/mechanism split is not as clearly defined in the Mach/RP3 family scheduler.

However, binding threads to processors is somewhat cumbersome under the CMU approach. Threads are bound to a processor by assigning them to a processor set that holds just one processor. A given processor cannot serve both as the host for one or more bound threads and as a member of the pool of processors available to the unbound threads of a family. The RP3 approach allows a processor to be used in this way because processor allocation and thread binding are separate operations.

- *Operating system support for the RP3 memory architecture*

The Mach operating system was designed for shared-memory multiprocessors, but it does not support nonuniform-memory-access (NUMA) architectures such as that of the RP3. A straightforward Mach implementation on the RP3 would have made all memory access consistent and essentially uniform by disabling the individual processor caches and by interleaving all virtual memory addresses across the entire machine. However, maximum RP3 performance could not have been achieved without exploiting the cache and local-memory features of the architecture, and exploiting these features required extensions to the Mach system.

A major decision we had to make early in the development was whether to expose the nonuniform memory model to application programs (that is, to let the programmer deal with such issues as memory placement and cacheability), or to preserve the standard Mach program interface while augmenting the operating system with algorithms that try to exploit the nonuniform memory architecture without help from the programmer. The latter approach has been investigated by the Platinum project at the University of Rochester [22] and by the 8CE project at IBM Research [23]. We chose the former approach for two reasons. First, it was consistent with the philosophy of making the basic hardware features of the architecture freely available to application programs and language run-time environments. Application and language researchers insisted on explicit control over memory placement and cacheability and did not want the operating system making such decisions for them. For these users, it was a requirement that we export the nonuniform memory model. Second, we believed that without help from application programmers or compilers, the operating system had little chance of making optimal placement and cacheability decisions. The 16KB RP3 page size was so large that unless an application segregated its data explicitly into regions with similar usage patterns, there was little hope that a given page would have such homogeneous content that a single choice of location and cacheability would be optimal. Once an application had partitioned its data into typed regions, it could easily indicate to the system the location and cacheability decisions that were appropriate for those regions.

While, in principle, the RP3 interleaving mechanism allowed the specification of different interleave amounts for different virtual pages, the mechanism was normally used to spread virtual pages across all 64 memory modules of RP3x. Supporting multiple interleave amounts would have significantly complicated the real-storage allocation algorithms in the operating system, and we never found an application whose requirement for this feature was significant enough to warrant the implementation effort. (We almost never ran more than one application at a time on RP3x, so using the interleaving mechanism to isolate independent jobs from one another was not attempted. See [3] for further discussion of this issue.) We simply partitioned the real memory of the machine into interleaved and noninterleaved pages at boot time. Regions of equal size (the size was a boot-time parameter) were taken from each PME and coalesced into a single region whose pages were interleaved across all the PMEs of the machine. We called this region the *global* region, although any particular byte of the global region was necessarily located in the local memory of some PME. Real memory

that was not incorporated in the global region remained in processor-specific *local* regions.

Virtual-memory attributes

The standard Mach programming interface included system calls that could be used to specify the protection level for pages of a virtual address space and to specify the manner in which such pages were to be inherited across UNIX *fork* operations. The Mach/RP3 interface added the notion of virtual memory *attributes* that a program could specify for pages of its virtual address space. A new system call, *vm_set_attributes*, let an application program place pages of its address space in memory of specified categories. Another new call, *vm_get_attributes*, let an application determine the current memory attributes of a particular page. The following virtual-memory attributes were defined:

- *Location attribute*. Virtual pages could be placed in global storage or in storage local to the processor to which a specified thread had been bound. Memory was declared local to a particular (bound) thread rather than to a particular processor, in keeping with the philosophy of not using processor identifiers in the system interface.
- *Replication attribute*. An application program could also request that specific virtual pages be copied from global storage into memory local to each bound thread that accessed those pages. Each bound thread essentially received a private copy of such pages. Unbound threads continued to share the original pages in global storage. The application program was responsible for keeping the replicated copies of a page consistent with one another and with the global copy, if such consistency was necessary. Replicated storage was typically used for read-only code pages.
- *Cacheability attribute*. Virtual pages could be made cacheable or noncacheable, independently of the location and replication attributes. All references to a cacheable page were handled through the local processor caches. Cache consistency was the responsibility of the application if a cacheable page was accessed by two or more threads.
- *Marked-data attribute*. A cacheable virtual page could be marked or unmarked (see the description of marked data in the RP3 hardware overview). Applications could use the marked-data attribute to improve the efficiency of software cache-coherence protocols.

By default, all application pages were placed in global storage and were made cacheable. This default value for the cacheability attribute was the wrong choice for parallel programs written for cache-coherent multiprocessors, but it let standard UNIX utility programs execute efficiently without modification. Single-threaded programs executed correctly in either cacheable or noncacheable mode on RP3, but they ran significantly faster in cacheable mode. Adding the necessary *vm_set_attributes* calls to the parallel programs that required noncacheable storage was not hard, because most such programs had to be modified for RP3 anyway.

Implementation

One of the important features of Mach is the separation of the memory-management system into machine-independent and machine-dependent layers. The machine-independent layer, which is called simply the VM module, implements all the traditional functions of a memory-management system. It manages the pool of free physical pages, it maintains mappings from user virtual address spaces to physical pages, and it keeps track of virtual pages that have been paged to disk. The machine-dependent layer is called the physical map, or *pmap*, module. The pmap module is responsible for maintaining actual hardware page tables, which are of course architecture-specific. It presents a well-defined procedural interface through which the VM module can request that particular virtual-to-physical mappings be entered in or removed from machine-specific page tables.

The physical address space that is used in the interface between the VM and pmap modules is really an abstraction maintained by the pmap module. It may or may not match the actual storage layout of the underlying hardware. This property of the interface allowed us to implement most of the support for the RP3 NUMA architecture in the machine-dependent pmap module. Our machine-independent VM module maintained attribute specifications for regions of address spaces and passed them to the pmap module when necessary, but it did not itself interpret the specifications. The pmap module was responsible for ensuring that mapped virtual pages were located in memory of the types specified by their attributes.

Implementation of the NUMA support for RP3 took place in two phases. In the first phase, the operating system itself was restructured to make use of the RP3 memory architecture. To complete this phase, we had to partition the operating system code and data structures into local and global segments. The local segment of the operating system consisted of processor-specific data structures required by the hardware (interrupt vectors and machine-state save areas, for example) and operating-system code that executed in real mode (first-level interrupt handlers, for example). Most of the operating-system code executed in virtual mode and was placed in the global segment, along with the bulk of the system data structures. The local segment was replicated in the local storage of each PME, while a single copy of the global segment was placed in global storage—that is, in storage

that was interleaved across all the PMEs. The resulting storage layout is shown in **Figure 5**.

In the second phase, the user-level system calls that let applications exploit the RP3 memory architecture were implemented. The different virtual-memory attributes required different implementation strategies.

Cacheability and marked-data attributes The implementation of the cacheability and marked-data attributes was straightforward. The cacheability of a virtual page was determined by a pair of bits in the page-table entry that mapped the virtual page to a physical page. When the pmap module was asked to enter a virtual-to-physical mapping in a page table, the required attributes for the virtual page were passed as part of the request. If "cacheable" or "marked-data" were among the attributes, the pmap module simply set the appropriate page-table-entry bits as it constructed the mapping.

Location attribute The implementation of the location attribute was more complicated. In Mach/RP3, the pmap module had to be able to move pages from one region of memory to another without involving the VM module, so it was necessary to implement a new level of indirection below the physical memory abstraction presented to the VM module. For the remainder of this discussion, we use the adjective "physical" to describe the abstract physical memory that is used in the interface between the pmap module and the VM module, and we use the adjective "absolute" to describe the true physical memory of the RP3 machine.

The abstract physical address space is illustrated in **Figure 6**. For clarity, the diagram shows just two PMEs and two virtual-address spaces. The machine-independent VM module maintained mappings from a number of virtual address spaces to the single physical address space, while the pmap module maintained a one-to-one mapping between the physical and absolute address spaces. From the figure, one can see how a page could be moved from global to local storage without changing the virtual-to-physical mappings maintained by the VM module. Of course, the actual page tables maintained by the pmap module were used by the hardware memory-management units and had to map virtual addresses directly to absolute addresses. Therefore the pmap module had to keep its page tables up-to-date when it relocated pages and permuted the physical-to-absolute mapping.

The pmap module constructed the initial physical-to-absolute mapping at boot time. Those physical pages that corresponded to global absolute pages were placed on the single machine-independent free-page list maintained by the VM module. The remaining physical pages were placed on local free lists that were maintained by the pmap module itself. Essentially, the VM module managed the

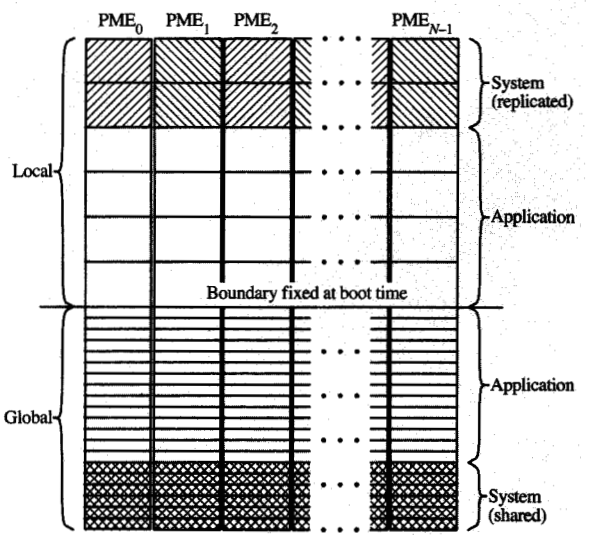

Figure 5

Operating system and application storage layout in Mach/RP3.

pool of pages that were interleaved across the entire machine, while the pmap module managed the pages that were local to individual processors.

The VM module did not interpret virtual-memory attributes and therefore did not distinguish local from global pages. Before a page could be used, however, a mapping for it would have to be entered in a machine-dependent page table; at that time the pmap module could relocate the page, if necessary, to make its location consistent with its attributes. When the pmap module was asked to establish a virtual-to-physical mapping, it would check the attributes specified for the virtual page against the current location of the specified physical page. If the physical page were in the wrong location for the specified attributes (global when it should have been local, local when it should have been global, or local to one processor when it should have been local to another), the pmap module would allocate an appropriate physical page, either from one of its local free lists or from the VM-module global free list. It would copy the contents of the original absolute page to the newly allocated absolute page and swap the physical-to-absolute mappings of the original and newly allocated physical pages. The original physical page would now be bound to an absolute page of the right type, and the pmap module could proceed to enter the mapping in a page table. The newly allocated physical page, which would now be bound to the original absolute page, would be returned to an appropriate free list.

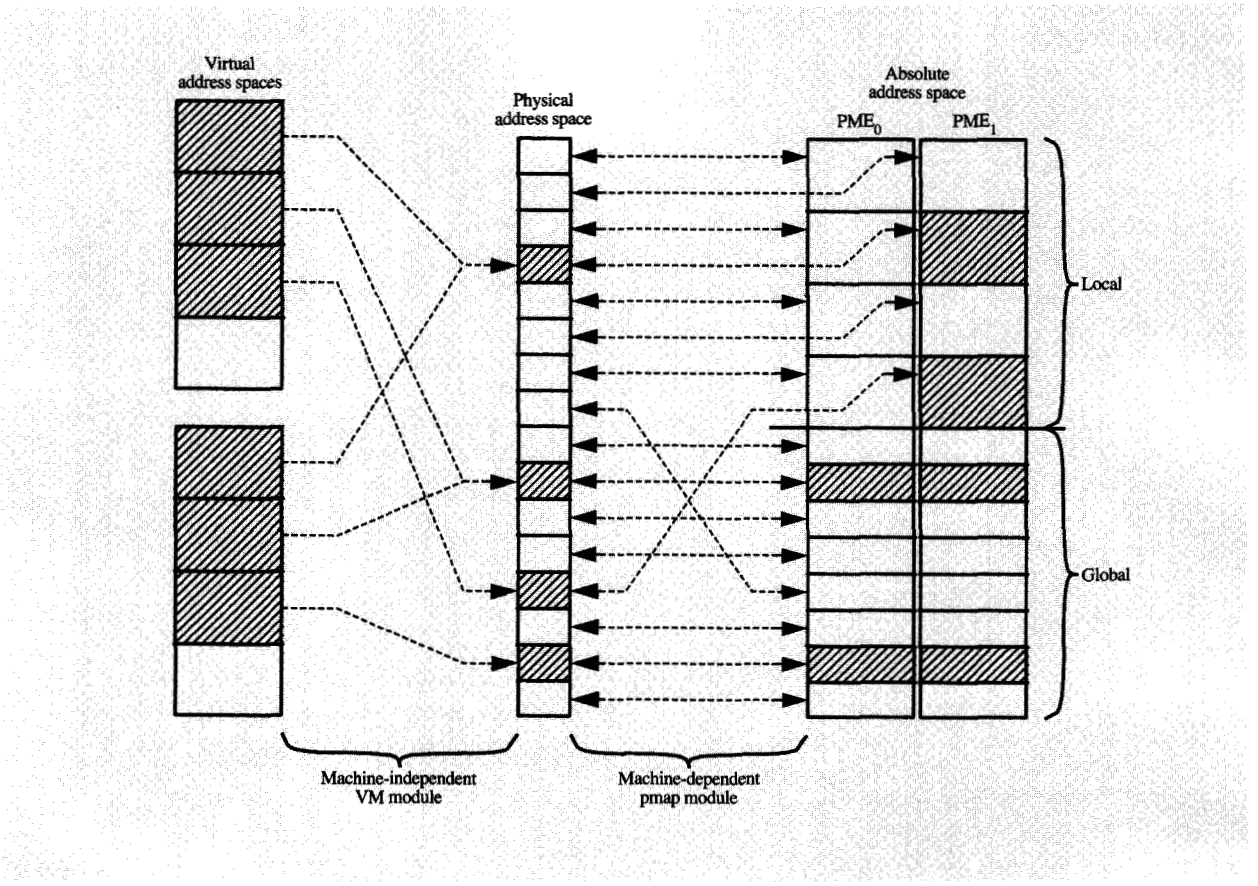

Figure 6

The abstract physical address space.

One problem with this approach was that since the VM module did not interpret memory attributes, it would frequently allocate and initialize a global page, only to have the page immediately copied to local memory. To eliminate this unnecessary copying, we extended the interface between the VM and pmap modules to let the pmap module allocate and deallocate local pages when appropriate.

Replication attribute The replication attribute required that a single virtual page be mapped to different absolute pages on different processors. The RP3 page-table structure, however, was designed to map each virtual page to a single absolute page. Since the structure was incapable of representing the one-to-many mapping implied by the replication attribute, we implemented a software extension of the page-table architecture. This extension was possible because the processors in RP3 had independent translation lookaside buffers and because system software could load mapping information directly into processor TLBs.

The Mach/RP3 pmap module maintained processor-specific virtual-to-absolute mappings in auxiliary tables associated with the page-table entries that would normally have mapped replicated virtual pages. These page-table entries were marked invalid, so that any attempt to access a replicated page would cause a page fault. When a fault occurred on a replicated page, the low-level page-fault handler would look in the auxiliary table for a processor-specific mapping for the page. If no such mapping existed, the fault would be passed to the VM module as an ordinary page fault. If a processor-specific mapping were found, the page-fault handler would load it directly into the processor TLB and would return immediately to the executing thread. Subsequent accesses to the replicated page would continue to succeed as long as the mapping remained in the TLB. Because each processor had its own TLB, the replicated virtual page could

be mapped to different absolute pages on different processors.

- *Operating system support for performance measurement on RP3*

Since RP3x was an experimental machine built to help users explore issues related to parallel processing, support for performance measurement was an important part of the operating system. It included the *virtual performance measurement chip* (VPMC) facility and the *event-logging* facility.

RP3 performance monitor

As discussed previously in the RP3 hardware overview, each PME on RP3 included a performance-measurement chip (PMC). The PMC contained a set of counters, a memory for samples of switch-delay times, and control and status registers. PME components would signal event occurrences to the PMC, and the PMC would increment the corresponding counters. A *master counter* in the PMC was incremented each processor cycle. The events counted by the PMC included

- Instructions completed.
- Floating-point operations completed.
- Translated memory requests.
- Nontranslated memory requests.
- Local-memory requests.
- Nonlocal-memory requests.
- Cacheable-memory requests.
- Cache misses.
- TLB misses.
- Memory requests from other PMEs.
- Memory-controller busy cycles.
- Switch-interface waiting cycles.

In addition to maintaining event counts, the PMC could sample switch-delay times. When any of the PMC counters overflowed, the PMC would freeze all the counters and, if the interrupt-enable bit were set in the PMC control register, would generate a processor interrupt.

The PMC was a device on the processor bus and was accessible from user programs. The advantages of using the PMC directly were low overhead and precise control, but there were problems as well. First, the PMC master counter was only 22 bits wide and overflowed every 1.2 seconds, so measuring a program section that executed longer than that interval was difficult. Second, any context switch that occurred during a monitored portion of a program would corrupt the measurements. Third, the PMC could record only 16 switch-delay samples. The virtual PMC facility was created to solve these problems.

Virtual PMC facility

A *virtual PMC*, or *VPMC*, was a system-maintained data structure that accumulated information from one or more physical PMCs and provided it to applications in a convenient form. A VPMC provided 64-bit counters that essentially never overflowed, and it provided a histogram of switch-delay samples that could contain an arbitrary number of entries. VPMCs could be associated with processors or with individual threads. A *processor VPMC* was like an actual PMC, in that it recorded all the activity on a particular PME. A *thread VPMC*, on the other hand, recorded just the activity of a particular thread. It was insensitive to context switches and continued to accumulate measurements as the thread moved from one processor to another. Processor and thread VPMCs could be active concurrently.

Mach/RP3 provided three methods for accessing a VPMC:

- *Mach-IPC access*. Mach/RP3 provided VPMC access as a service available through the general Mach interprocess communication (IPC) facility. An application could use this service to obtain a current copy of any VPMC for which it held the necessary access rights. (Controlled distribution of access rights is a feature of Mach IPC.) The Mach IPC mechanism made VPMC access available both to applications running on RP3x itself and to monitoring programs running outside RP3x.
- *System-call access*. Mach/RP3 provided a system call that would return a current copy of the calling thread VPMC. This method lacked the generality of the Mach IPC version, but it ran much faster.
- *Memory-mapped access*. Mach/RP3 provided a mechanism for mapping a VPMC data structure directly into an application address space, where it could be examined without operating system intervention. The information in a mapped VPMC could be accessed very quickly, but it was usually somewhat out-of-date because it was updated only when the underlying hardware PMC generated an interrupt. A thread could obtain up-to-date information about its own activity by reading the hardware PMC directly and adding the counter values to the corresponding counters of its own mapped VPMC. A concurrent readers/writers synchronization algorithm was used to avoid reading the mapped data while the operating system was updating the VPMC.

The costs (measured in numbers of instructions) of the various PMC and VPMC access methods are shown in **Table 1**. The programmer could choose the cheapest access method that satisfied the functional requirements of a particular application. Further details on the RP3 VPMC facility can be found in [24].

Table 1 Number of instructions required for PMC and VPMC access.

Access method	*Number of instructions*
Direct PMC	50
Memory-mapped VPMC	500
System-call VPMC	3000
Mach-IPC VPMC	14,000

Event-logging facility

The performance of a parallel program on RP3 depended on the performance of many components, including the program itself, libraries the program used, the operating system, and the hardware. One method for determining where time was spent was to instrument the various components in order to collect traces. The RP3 *event-logging* facility was developed to provide a uniform means for recording trace information from all levels of the system.

An *event record* consisted of a time stamp, a processor number, an integer event type, and a variable-length data section. The event-logging facility maintained a separate event buffer or log in the local storage of each PME. Events generated on a particular PME were recorded in its log, so event logging itself did not increase switch traffic. Events from different logs could later be correlated by time stamp. A boot-time parameter determined the size of the local event logs. Events could be generated by the operating system and by application programs, and an application could read and remove event records from the local event logs. An event record was discarded if the local log was full when the event was generated, but the fact that events were lost was itself recorded as an event.

The RP3 event-logging facility was used to collect data from all components of the RP3 system. The operating system could be configured to record hardware PMC data in the event log. Operating system events such as scheduling points and page faults could also be recorded. The C-Threads package [13] was instrumented to generate events corresponding to internal scheduling decisions. Finally, a variety of application programs were modified to generate application-specific events. A programmer could use the event-logging facility to correlate the moment-by-moment behavior of a parallel program with activity in the underlying run-time package, operating system, and hardware. The RP3 performance visualization system [25] was developed to display data from the event-logging facility in a variety of graphical formats.

Concluding remarks

In this paper, we have discussed how the Mach operating system was extended to enable efficient parallel processing on RP3. Our goal was to provide an extended UNIX programming interface that allowed application code running on RP3 to take full advantage of the underlying processor and memory structure. In addition, we provided convenient facilities for measuring the performance of applications running on the machine, and we provided a simple event-logging facility that allowed measurement data from multiple sources in the system to be combined and accessed in a uniform way.

Our philosophy of allowing the user full access to the basic facilities of the hardware ran counter to the prevalent operating system philosophy of hiding the complexities of the underlying machine from the user. For the range of applications that RP3 was designed to investigate, we believe the usual approach was incorrect and would not have allowed the maximum performance to be achieved. Our experience has been that compiler and run-time-package developers wholeheartedly supported our approach, while users who had to deal directly with the system-call interface found the system difficult to use. This was to be expected, since our decisions were made in favor of those programmers and compiler developers willing to expend significant effort in exploiting the RP3 architecture. The long-range intent was that application programmers would not have to deal directly with the family-scheduler or memory-attributes system calls, but would instead deal with the problems of specifying parallel algorithms in forms acceptable to compilers that used these interfaces.

Acknowledgments

The RP3 project was supported in part by the Defense Advanced Research Projects Agency under Contract Number N00039-87-C-0122 (Multi-processor System Architecture). Our work on RP3 depended on the assistance of many people, without whose conscientious efforts the system would never have existed. In particular, we would like to thank Frances Allen, Gordon Braudaway, Walter Kleinfelder, Matthew Thoennes, and Herbert Liberman for their assistance and support during the RP3 project. Rajat Datta implemented the System/370 side of the RP3 I/O system; Rajat also contributed significantly to the first port of Mach to an RP3 PME. Anthony Bolmarcich was responsible for the implementations of EPEX for Mach/RT and Mach/RP3. The original port of Mach to the RP3 simulator was completed by Daniel Julin of Carnegie Mellon University while he was a summer student working in the IBM Research Division. The Mach research group at Carnegie Mellon University, led by Professor Richard Rashid, was very supportive of our work with Mach, and their special efforts to make the latest versions of Mach available to us are greatly appreciated. Daniel Rencewicz of the IBM-CMU Information Technology Center was responsible for obtaining production releases of Mach for the RT® system and making them available within IBM.

References

1. S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Co., Reading, MA, 1989.
2. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Conference*, USENIX Association, Atlanta, July 1986, pp. 93–113.
3. R. Bryant, H. Chang, and B. Rosenburg, "Experience Developing the RP3 Operating System," *Computing Syst.* **4**, No. 3, 183–216 (1991).
4. R. Rettberg, W. Crowther, P. Carvey, and R. Tomlinson, "The Monarch Parallel Processor Hardware Design," *IEEE Computer* **23**, No. 4, 18–30 (1990).
5. P. Yew, "Architecture of the Cedar Parallel Supercomputer," *CSRD Technical Report 609*, Center for Supercomputing Research and Development, University of Illinois, Urbana–Champaign, August 1986.
6. A. Gottlieb, B. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Trans. Programming Lang. & Syst.* **5**, No. 2, 164–189 (1983).
7. G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Computer Society, Chicago, August 1985, pp. 764–771.
8. W. Brantley, K. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference on Parallel Processing*, IEEE Computer Society, Chicago, August 1985, pp. 782–789.
9. F. Darema, D. George, V. Norton, and G. Pfister, "A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN," *Parallel Computing* **7**, No. 1, 11–24 (1988).
10. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *J. Parallel & Dist. Computing* **5**, No. 5, 617–640 (1988).
11. V. Sarkar, "Automatic Partitioning of a Program Dependence Graph into Parallel Tasks," *IBM J. Res. Develop.* **35**, 779–804 (1991, this issue).
12. S. F. Hummel and E. Schonberg, "Low-Overhead Scheduling of Nested Parallelism," *IBM J. Res. Develop.* **35**, 743–765 (1991, this issue).
13. E. Cooper and R. Draves, "C Threads," *Technical Report CMU-CS-88-154*, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1988.
14. T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, Pacific Grove, CA, October 1991, pp. 95–109.
15. A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, Pacific Grove, CA, October 1991, pp. 41–55.
16. S. Leutenegger and M. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proceedings of the ACM Sigmetrics Conference*, Boulder, CO, May 1990, pp. 226–236.
17. J. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proceedings of the Distributed Computing Systems Conference*, IEEE Computer Society, 1982, pp. 22–30.
18. M. Seager and J. Stichnoth, "Simulating the Scheduling of Parallel Supercomputer Applications," *Technical Report UCRL-102059*, Lawrence Livermore National Laboratory, Livermore, CA, September 1989.
19. A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, Litchfield Park, AZ, December 1989, pp. 159–166.
20. B. Rosenburg and H. Chang, " 'Ration Function' that Lets a Parallel Program Adapt Its Processor Requirements to System Load," *IBM Tech. Disclosure Bull.* **32**, No. 8B, 123–125 (1990).
21. D. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer* **23**, No. 5, 35–43 (1990).
22. A. Cox, R. Fowler, and M. Scott, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, Litchfield Park, AZ, December 1989, pp. 32–44.
23. W. Bolosky, R. Fitzgerald, and M. Scott, "Simple But Effective Techniques for NUMA Memory Management," *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, Litchfield Park, AZ, December 1989, pp. 19–31.
24. H. Chang and W. Brantley, "Support Environment for the RP3 Performance Monitor," *Proceedings of the Workshop on Supercomputer Measurement and Visualization*, Addison-Wesley Publishing Co., Reading, MA, 1990, pp. 117–134.
25. D. N. Kimelman and T. A. Ngo, "The RP3 Program Visualization Environment," *IBM J. Res. Develop.* **35**, 635–651 (1991, this issue).

Received November 2, 1990; accepted for publication April 19, 1991

Raymond M. Bryant *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Bryant received his B.S. degree from the South Dakota School of Mines and Technology, Rapid City, in 1971, and his Ph.D. in 1978 from the University of Maryland, College Park, both in mathematics. From 1978 to 1981 he was on the faculty of the Computer Sciences Department at the University of Wisconsin–Madison. He joined IBM in 1981 as a Research Staff Member in the Computer Sciences Department of the IBM Thomas J. Watson Research Center. He began

working with the RP3 project in 1985, and in 1989 received an IBM Outstanding Technical Achievement Award for his work on a functional simulator for RP3. Since 1991, he has been working on the use of shared virtual memory for parallel processing and on operating systems for multicomputers. Dr. Bryant is a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers.

Hung-Yang Chang *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Chang received his Ph.D. in computer sciences from the University of Wisconsin–Madison in 1987. His Ph.D. dissertation was a study of distributed real-time scheduling algorithms. He has been a Research Staff Member in the Computer Sciences Department of the IBM Thomas J. Watson Research Center since 1987. Dr. Chang initially joined the operating system group of the RP3 multiprocessor project. He is currently interested in extending microkernel operating systems to support parallel programming and in global job scheduling (load distribution). Dr. Chang is a member of the Association for Computing Machinery.

Bryan S. Rosenburg *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Rosenburg received his B.S. degree in mathematics from Michigan State University in 1979, and his Ph.D. in computer sciences from the University of Wisconsin–Madison in 1986. In 1987 he joined the IBM Thomas J. Watson Research Center as a Research Staff Member with the operating system group of the RP3 project. His current research interest is in exploring microkernel-based operating systems for large-scale non-shared-memory multicomputers. Dr. Rosenburg is a member of the Association for Computing Machinery.