Experience Developing the RP3 Operating System[1]

Ray Bryant, Hung-Yang Chang, Bryan Rosenburg
IBM Thomas J. Watson Research Center

ABSTRACT: RP3, or Research Parallel Processing Prototype, was the name given to the architecture of a research vehicle for exploring the hardware and software aspects of highly parallel computation. RP3 was a shared-memory machine designed to be scalable to 512 processors; a 64 processor machine was in operation for two and half years starting in October 1988. The operating system for RP3 was a version of the Mach system from Carnegie Mellon University. This paper discusses what we learned about developing operating systems for shared-memory parallel machines such as RP3 and includes recommendations on how we feel such systems should and should not be structured. We also evaluate the architectural features of RP3 from the viewpoint of our use of the machine. Finally, we include some recommendations for others who endeavor to build similar prototype or product machines.

1. Supported in part by the Defense Advanced Research Projects Agency under Contract Number N00039-87-C-0122 (Multi-processor System Architecture).

1. Introduction

The RP3 project of the IBM Research Division had as its goal the development of a research vehicle for exploring all aspects of highly parallel computation. RP3 was a shared-memory machine designed to be scalable to 512 processors; a 64-way machine was built and was in operation from October 1988 through March 1991.

The authors of this paper were responsible for creating the operating system environment used to run programs on RP3. (The operating system for RP3 was a version of Mach [2], which is a restructured version of BSD 4.3 Unix.) The extensions we made to Mach to support RP3 are described in [7] and will not be discussed in detail here. Instead, this paper summarizes our experience developing Mach/RP3 and presents our views on how operating systems for highly parallel shared-memory machines such as RP3 should be constructed, as well as our experience in supporting and using this system for parallel processing research.

In the following sections of this paper, we provide an overview of the RP3 architecture and a brief history of the RP3 project. We then discuss the lessons we feel we learned during the course of this project and we make some recommendations to developers of similar machines.

2. RP3 Hardware Overview

Figure 1 illustrates the RP3 architecture. An RP3 machine could consist of up to 512 *processor memory elements* or PME's. The prototype hardware that was actually built, which we will refer to as RP3x, consists of 64 PME's. Each PME included the following components:

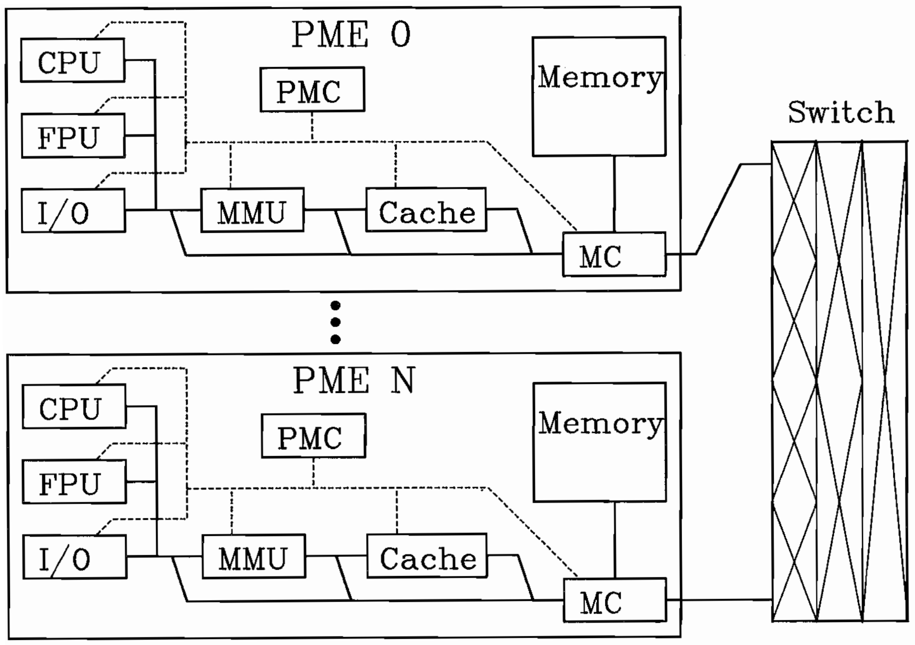

Figure 1: The RP3 Architecture

CPU The central processing unit, a 32-bit RISC processor known as the ROMP. The same processor was used in the original IBM RT workstation.

FPU The floating point unit, similar to the floating point unit found in second-generation IBM RT workstations. It used the Motorola MC68881 floating point chip which implements the IEEE floating point standard.

I/O The I/O interface, which provided a connection to an IBM PC/AT that served as an I/O and Support Processor, or ISP. Each ISP was connected to 8 PME's and to an IBM System/370 mainframe.

MMU The memory management unit. The MMU supported a typical segment and page table address translation mechanism and included a 64-entry, 2-way set-associative translation lookaside buffer (TLB).

CACHE A 32-kilobyte, 2-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 kilobytes.

The 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 top 9 bits of the address specified the target PME.

A 1- to 8-megabyte memory module. (The 64-way prototype, RP3x, was fully populated with 8-megabyte memory modules.) Note that all memory in RP3 was packaged with the processors.

The 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.

All the PME's of an RP3 machine were connected by a multistage interconnection network or switch. The switch, which was constructed of water-cooled bipolar technology, had 64-bit data paths and a bandwidth of roughly 14 megabytes/second per PME. All memory on RP3 was local to individual PME's but was accessible from any processor in the machine. However, a performance penalty was incurred when accessing remote memory. RP3x had an access time ratio of 1:12:20 between cache, local, and remote memory, assuming no network 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* or *NUMA* machines. Support of this NUMA architecture required operating system extensions that are discussed in [7].

To avoid potential memory bottlenecks, the RP3 memory management unit supported the concept of software-controlled *interleaved* memory. Addresses for interleaved memory underwent an additional transformation after virtual to real address translation. The interleaving transformation exchanged bits in the lowand highorder portions of the real address (see Figure 2). 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 doublewords being stored in different memory modules. The number of bits interchanged (and hence the base 2 logarithm of the number of modules used) was specified by the interleave amount in the page table. Figure 2 shows how the interleaving transformation could be used to spread virtual pages across multiple PME's.

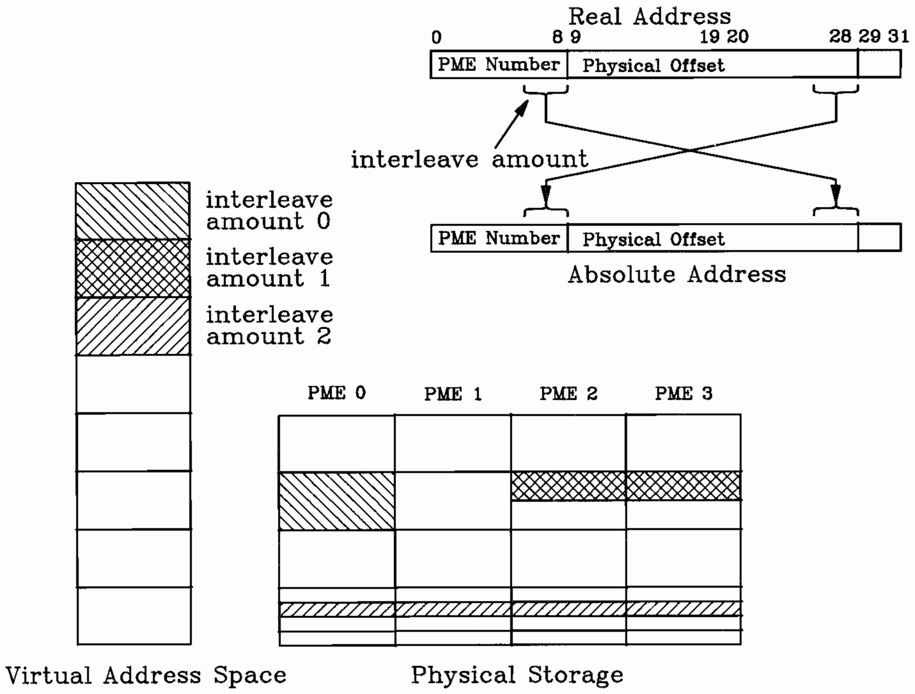

Figure 2: The RP3 interleaving transformation

Normally, all data used by more than one processor was stored in interleaved memory. For this reason, interleaved memory is also referred to as *global* memory. Local, or non-interleaved, memory is referred to as *sequential* memory.

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 RP3 hardware did not provide any mechanism for keeping caches coherent between PME's; cache coherency had to be maintained in software. The cache was visible to application code in the sense that user-mode instructions to invalidate the cache were provided. In addition, the page tables included cacheability information, so that ranges of virtual addresses (on page boundaries) could be specified as cacheable or non-cacheable. Since there was no page table associated with real-mode memory access, all real-mode memory accesses on RP3 were non-cacheable references. 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 supported the *fetch&add* [10] 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 [1] and [6]. RP3x, our working 64-way prototype, differed from the published design in the following respects:

- The I/O and Support Processors, or ISP's, in RP3x were simple IBM PC/AT's rather than the elaborate custom-built machines described in the published RP3 design. Each PC/AT was connected to 8 PME's and could access RP3 memory through its PME's. Memory requests from an ISP were identical to requests from a PME's own processor, so an ISP could address real or virtual memory that was local, remote, or even interleaved. The ISP-PME bandwidth was roughly 500 kilobytes/second and could be dedicated to a single PME or multiplexed among PME's. An ISP could raise an interrupt in any of its PME's, and a PME could signal its ISP. The I/O hardware allowed such a signal to interrupt the ISP, but our ISP software was synchronous and periodically polled each PME's signal line.
  In the original RP3 design, each ISP was to be directly connected to devices such as disks and networks. In the implemented design, the ISP's were channel-connected to a System/370 mainframe which in turn could access a large collection of disks and other devices. Bandwidth between an ISP and the System/370 was roughly 3 megabytes/second. RP3 I/O requests were passed from a PME to its ISP to the System/370 and back.
- The original RP3 design called for a *combining switch* that would reduce memory and switch contention by merging *fetch&op* operations when they collided at interior switch elements. The design could not be implemented in the technology available at the time. RP3x supported the full range of *fetch&op*'s, but the operations were serialized at individual memory controllers and were not combined in the switch.

- The floating point processors in RP3x were based on a standard RT workstation floating point unit and incorporated Motorola MC68881 floating point chips implementing the IEEE floating point standard. The original RP3 design called for vector floating point processors implementing the System/370 floating point standard.
- The RP3x cache system limited the PME clock rate to 7 MHz, about a third of the originally projected rate. Furthermore, the RP3x memory controller could support just one outstanding request to the memory subsystem at a time rather than the eight outstanding requests it was designed to handle.
- The RP3x memory management unit did not support hardware reload of the translation lookaside buffer (TLB). When a processor made a memory request to a virtual address that was not mapped by the TLB, an exception was raised, and a software exception handler had to explicitly load translation information for the faulting address into the TLB.

3. *History of RP3*

Our experience with RP3 is closely related to its development history, so it is useful to summarize the major milestones of this project. The original idea that the IBM Research Division should attempt to build a large parallel processor apparently originated with a task force led by George Almasi during the winter of 1982-83. The earliest the name *RP3* was actually used appears to be in the fall of 1983, when a group led by Greg Pfister began to design the RP3 architecture. In October of 1984, the IBM *Corporate Management Committee* agreed to fund the RP3 project.

With funding secured, the existing project team was expanded to include a design automation group, a processor design group, a technology group (responsible for constructing the machine and coordinating production of parts with the IBM development divisions), and a software development group. The software group initially concentrated on the development of parallel applications. Since no parallel hardware was available, the approach selected was to emulate a virtual parallel processor under VM/370. This effort produced the EPEX [8] system, and a significant library of applications were written using the EPEX parallel programming extensions to Fortran.

Other significant technical milestones:

Dec. 1984 RP3 architecture frozen. With the exceptions noted previously, this architecture was an accurate description of the 64-way prototype, RP3x.

Aug. 1985 A set of papers on RP3 were published in the *Proceedings of the 1985 International Conference on Parallel Processing* [1,6,16].

Dec. 1985 Power/mechanical frame completed and installed in lab.

Jun. 1986 Uniprocessor version of RP3 instruction-level simulator completed.

Aug. 1986 First version of Mach on RP3 simulator completed.

Dec. 1986 First complete processor chip set assembled and tested.

Apr. 1987 Final-pass chip designs released to manufacturing.

Sep. 1987 EPEX environment ported to Mach/RT.

Sep. 1987 First full PME with final-pass chips completed.

Sep. 1987 Multiprocessor version of RP3 instruction-level simulator completed.

Oct. 1987 Mach/RP3 runs on first PME.

Nov. 1987 Mach/RP3 runs on multiprocessor RP3 simulator.

Feb. 1988 Mach/RP3 runs on 2-processor hardware.

Jun. 1988 Mach and three EPEX test applications run on 4-processor hardware.

Aug. 1988 Mach and three EPEX test applications run on 8-processor hardware.

Oct. 1988 64-processor prototype (RP3x) completed and turned over to software team.

Nov. 1988 64-way application speedup experiments completed on three EPEX test programs.

Feb. 1989 Mach/RP3 with cacheability control and interleaved memory support completed.

Mar. 1989 Mach/RP3 with processor allocation primitives and local memory support completed.

Jun. 1989 RP3x upgraded to include cache and PMC.

Jul. 1989 RP3x available to outside users via NSF net.

Mar. 1990 RP3x upgraded with floating point coprocessors. (Previously, all floating point operations were emulated in software.)

Mar. 1991 RP3x decommissioned.

A final historical note: all the authors of this paper joined the RP3 project after the initial design for the machine was complete. Thus we are unable to comment on the design process that led to the RP3 architecture, but only on the results of that process.

4. *Lessons Learned from RP3 Operating Systems Development*

We feel we learned a great deal during the years we were part of the RP3 project, not only about the RP3 architecture itself, but also about the art of programming such machines, and about the difficulty of managing such a large research project. The rest of this paper presents some of the things we learned, categorized roughly into three areas: evaluations of specific RP3 architectural features, discussions of a variety of software development issues for such machines, and general comments about large research projects and the future of sharedmemory multiprocessing.

5. *RP3 Architecture*

5.1 Interleaved Memory

The RP3 programmable interleaving mechanism let us take advantage of both the performance of local memory and the convenience of a nearly uniform global shared memory. However, the mechanism was too inflexible to be used easily in other than trivial ways.

Globally interleaved memory let us program an RP3 machine as a true shared-memory multiprocessor. We were able to run RP3x using a single copy of the kernel text and data; partitioning of kernel data structures on a per-PME basis was not required. Furthermore, optimizations such as copy-on-write and shared user text segments were feasible in interleaved memory. Use of these optimizations in sequential memory could have resulted in intolerable memory contention.

Early versions of Mach/RP3 did not support interleaved memory. All kernel text and static data resided in a single memory module, and most application programs were small enough to fit in a few 16-kilobyte pages. For these versions of the kernel, we found memory and network contention to be significant, especially the contention caused by instruction fetches. (In particular, instruction fetches from the busy-wait loop in the kernel *simple_lock* routine caused kernel startup on the 64-processor prototype to take an inordinate amount of time.) Furthermore, text and read-only-data regions of related tasks were shared copy-on-write. This optimization reduced memory usage and copying costs, but it caused severe contention for the memory modules in which the resulting shared pages resided. Application programmers found it necessary to artificially modify the read-only regions of their programs to force them to be replicated.

These problems were alleviated when we restructured the Mach/RP3 machine-dependent memory management subsystem to support the RP3 interleaving mechanism. Without interleaving, we would have been forced to replicate the kernel text to every processor and to disable the Mach kernel optimizations that result in user-level memory sharing, and we would have had to devote a considerable effort to partitioning the kernel data structures into PME-specific regions.

Our early memory contention problems were exacerbated by the initial lack of processor caches in RP3x. Processor caches can alleviate some of the contention caused by instruction fetches and accesses to read-only or private data. Nevertheless, interleaving of non-cached data structures was still important, and even for cached pages, interleaving increased the bandwidth available for cache reload.

While globally interleaved memory was important on RP3, also important was the ability to use local or non-interleaved memory. The access time to local memory was half that to remote memory, even without switch or remote memory contention. Without processor caches, the use of local memory would have been absolutely critical. With the write-through caches of RP3, the use of local memory was important because it kept memory writes off the switch.

The RP3 programmable interleaving mechanism was important because it provided both globally interleaved and local memory, but we found it was too inflexible to make intermediate interleaving levels useful. The interleaving transformation could spread a virtual page across a set of PME's smaller than the entire machine, but it could not spread a virtual page across an arbitrary subset of the machine. The interleave amount in the page table was the base 2 logarithm of the number of memory modules used, and the page was interleaved across a range of adjacent memory modules that began at some multiple of that amount. For example, if the interleave amount was 4, the page was spread across 16 consecutive memory modules, starting with PME 0, 16, 32, 48, etc. These restrictions made it difficult to interleave pages across just those processors allocated to a particular parallel application. Furthermore, an interleaved page sliced across the real memory of the machine and occupied part of a page frame in each memory module. The rest of the memory in the affected page frames could only be used in identically interleaved virtual pages. Supporting variable interleave amounts would have made the allocation of real memory a two-dimensional bin packing problem. Also, given the presence of processor caches, it seemed most appropriate to use the maximum interleave amount so as to make the maximum memory bandwidth available for cache reload. For our purposes, a single page table bit indicating whether a page was located in sequential memory or was interleaved across the entire machine would have been sufficient.

The variable interleave mechanism could have been used to statically partition an RP3 machine, an option we will discuss in a later section.

5.2 Software-Controlled Caches

Local caches proved to be critical to RP3 performance even though the caches were not kept consistent across processors. However, the potential for a software-controlled cache to be as efficient as a hardware-controlled cache was not demonstrated by RP3.

Coherence was not an issue for pages that were used by just one processor, or pages that were read-only. Cacheing such pages was very important, even for applications that made heavy use of shared read-write memory.

Nevertheless, the RP3 cache included features designed to let sophisticated applications explicitly manage cached, shared-writable memory, but no compiler that could exploit the RP3 software-controlled cache was ever completed. Without compiler support, cache invalidation and software cache coherency were difficult to implement efficiently. To ensure safety, hand-coded software cache coherency schemes were forced to invalidate the cache too frequently and processor performance was dominated by cache cold-start effects. The RP3 "marked data" mechanism helped alleviate this problem but did not eliminate it.

Experiments indicated that, even if RP3 had hardware cache-coherency, it would often have been advantageous to keep shared variables in non-cacheable memory. Shared data was often read once and not reused quickly, and loading it into the cache could evict data that could profitably have been retained. This effect was particularly severe on RP3 because instructions and data shared a single cache.

The difficulty of dealing with RP3's non-coherent cache structure led us to execute the Mach kernel code with instructions and stack in cacheable memory, and all other data in non-cacheable memory. As a result, kernel code executed half as fast as user code that placed all data in cacheable memory. A hardware-coherent cache would have allowed us to place all kernel data in cacheable memory. The performance penalty for supporting a hardware-coherent cache would probably have been no worse than the performance penalty we incurred due to data being non-cacheable. Furthermore, a hardware-coherent cache would have allowed the use of the cache when the processor was executing in real mode. On RP3, all real-mode accesses were non-cacheable. While very little code on Mach/RP3 executed in real mode, the performance penalty of executing in non-cacheable mode was high and was one of the reasons we encountered significant performance problems related to TLB thrashing (see below).

We now believe multiprocessor architectures should support both coherent and non-coherent modes of execution. The overhead of hardware cache coherence protocols on highly-parallel machines may be unacceptably high for some applications. Code (such as the Unix kernel) that is too difficult to restructure to allow the use of software-level cache coherency would be executed in coherent mode. Restructured software that can gain from the reduced hardware overhead would execute in non-coherent mode.

5.3 Performance Monitor Chip

The RP3 performance monitor chip was an excellent facility for understanding RP3 system performance. It was also a relatively simple chip to implement, since it was mostly made up of counters and interface logic that let the processor read and reset the counters. The hard part of the design was in identifying and routing the signals from other parts of the PME to the PMC. In effect the PMC was a small per-processor hardware monitor. We used the PMC to identify the TLB-thrashing problem discussed below, to measure the latency of the global memory in the presence of contention, to count local and global memory references, and to observe cache utilization. We thus found the PMC to be crucial to the understanding of parallel system and application performance on RP3x.

As processors are built of denser and denser integrated logic, it becomes more and more difficult to obtain this kind of information by any other means. As integration densities continue to increase, it also becomes more feasible to allocate a few circuits for performance measurement. We therefore recommend that such instrumentation be included as a standard part of future processors.

5.4 Software TLB Loading

TLB thrashing can be a problem for systems using software TLB reload. On RP3, the translation lookaside buffer, or TLB, was a 2-way set-associative cache of virtual to real address translations. As usual, this cache was used to avoid the penalty of inspecting the page table during each virtual address translation cycle. On the 64-way prototype, TLB reload was performed in software at an expense of about 200 instructions per entry reloaded. Since these instructions had to be executed in real mode, instructions and data were not cached and therefore a TLB reload required approximately 2 milliseconds processing time. As a result, when TLB thrashing occured, it had a significant effect on program performance. This problem came to our attention when a user informed us that a parallel loop of 8 instructions took more than 100 times as long to execute on one of the 64 processors in the system as on the others. Our initial guess was that the processor in question had a hardware problem, but subsequent runs showed the problem moved from processor to processor in the system. Through use of the PMC we were able to determine that the slow processor was taking an enormous number of TLB misses (and hence reloads). It happened that the loop was so constructed that one of the 64 processors involved was trying to address instructions, local data, and global data, all using the same TLB entry. Since the TLB is only 2-way set-associative, each pass through the 8 instruction loop was causing three TLB misses, expanding the effective size of the loop from 8 to over 600 instructions. In general, we would recommend at least a 4-way set-associative TLB be used if software TLB reload is proposed for a particular machine.

5.5 Hot-Spot Contention

Hot spot contention, as defined in [16], was not a problem on RP3x. Contrary to previous predictions of "hot-spots" on RP3, our measurements showed insignificant delay due to hot-spot tree saturation on the 64-way prototype. In [18] Thomas reports that tree saturation is also not a problem on the BBN Butterfly, because the Butterfly switch is *non-blocking*. The RP3 switch is *blocking*, so for us the discrepancy between prediction and measurement has a different explanation. The original RP3 design, on which the tree saturation prediction was based, allowed up to 8 outstanding memory requests per processor. The RP3x prototype allowed only one outstanding request per PME. This engineering change made RP3x a processorand memory-bound machine, not a communicationbandwidth-bound machine.[3] Indeed RP3x could be regarded as a 64-processor machine with a 128-port global memory. 64 of the ports were dedicated to the local processors, leaving 64 ports to satisfy global memory requests. In order to avoid saturating a memory module, the pattern of memory references thus had to be nearly uniform since there was no extra capacity available to deal with an imbalance. In hindsight, it would have been better to construct the machine with 256 or more ports to global memory so that when the memory reference pattern was not completely uniform, extra bandwidth would have been available to service the requests at the more heavily-loaded memory modules.

6. Software Development

6.1 Functional Simulation

Timely development of the RP3 operating system would have been impossible without functional simulation of the RP3 architecture. Early in the project we obtained an instruction-level simulator of the RT workstation, and we were able to convert it first to a uniprocessor RP3 simulator and then to a multiprocessor simulator. The time invested in this effort was considerable, but it was time well spent because it let us develop system software well ahead of the hardware development

3. This change also eliminated the need for the RP3 fence-registers [6] and made RP3x a serially consistent machine with respect to store order.

schedule. The first version of Mach/RP3 was available under the RP3 simulator more than a year before the first prototype PME was completed. The Mach/RP3 kernel came to be regarded as the final architectural verification test for each new version of the prototype hardware. Without the simulator, we would never have had the confidence in the correctness of Mach/RP3 to use the kernel for this purpose.

The RP3 PME development plan required the custom design of several chips. Chip design "front loads" the processor development cycle in the sense that for the first two-thirds of the development cycle no prototype hardware is available. Once a set of correct chips has been completed, production of the parallel machine occurs at an accelerated rate. For the RP3 project, the first functioning PME was available two years after the project started; RP3x was completed approximately one year later. Without the RP3 functional simulator, we would probably not have completed the Mach/RP3 kernel until months after RP3x was built. Using the simulator, we were able to verify not only that the kernel was correct for the target architecture, but that the applications would execute correctly as well. We were ready to execute kernel and applications code on the 4-, 8-, and 64-way machines as soon as they were available. Without this ability, we would have fallen far behind the accelerated hardware schedule during the last year of the hardware development cycle.

Even if the PME design cycle had been much shorter, the simulator would have been valuable because it provided a much more productive development environment than any prototype hardware.

We found the conversion of a uniprocessor ROMP simulator to a simulator of RP3 to be a detailed, but relatively straightforward process. Aside from architectural changes between the ROMP and the RP3 PME, the hardest part of the problem was simulating the multiprocessor. Rather than rewrite the simulator, we converted it to a parallel program by replicating the simulator code in multiple virtual machines under VM/370, and by using memory shared between the virtual machines to represent RP3 memory. Since the simulator itself was run on a multiprocessor System/370, we were able to find and eliminate many multiprocessor timing bugs before the multiprocessor RP3 hardware was available. If we had run the simulator on a uniprocessor system, interactions between simulated processors would have been limited by VM/370 dispatching intervals, and we probably would not have found as many timing bugs.

6.2 The Mach Operating System

The Mach system from CMU was an excellent starting point for the RP3 operating system. The original plans for RP3 included a contract with the Ultracomputer project at New York University for the development of a Unix-compatible RP3 operating system based on the Ultracomputer Symunix operating system [9]. For a variety of reasons, our group chose to pursue Mach, first as an alternative, and then as the primary operating system for RP3. The selection of Mach as the basis for the RP3 operating system was a successful strategy for the following reasons:

- It allowed us to use the same operating system on RT workstations, on VM/370, and on RP3. These versions of Mach cooperated in supporting compilation, debugging, and testing of user code on RP3. The same programming environments and compilers executed under Mach/RT as under Mach/RP3, and users were able to accomplish much of their debugging on Mach/RT before moving to the parallel machine.
- It enabled the rapid development of an initial uniprocessor RP3 operating system. Mach was designed to be portable and maintains a fairly clear separation of machine-independent from machine-dependent code. Since RP3 used the same processor as the RT workstation, porting Mach/RT to a single RP3 PME was straightforward. Uniprocessor Mach/RP3 used not only the machine-independent code from Mach/RT but much of the machine-dependent code as well. The bulk of the porting effort involved memory management, because the RP3 and RT memory management units were radically different. Here again the porting effort was aided by Mach's clear encapsulation of machine-dependent memory management code.
- It aided the transformation of the uniprocessor RP3 operating system to a multiprocessor operating system. The machine-independent Mach code was multiprocessor-capable to begin with. We could concentrate on making the RT-based machine-dependent code multiprocessor-capable as well. In this effort we were aided by the examples provided by existing Mach implementations for a variety of commercial multiprocessors.

- It simplified the support of the RP3 memory architecture. Changes for global and local memory support as well as for user-level cacheability control were isolated in the machine-dependent portion of the kernel.

Some disadvantages of using Mach were that

- Large parts of the Mach kernel we used were globally serialized. This issue is discussed in more detail in the next section.
- Mach was not designed for NUMA multiprocessors. Adding support for the RP3 NUMA architecture was a major effort, albeit an effort that was aided by the modular design of the Mach memory management subsystem.
- Given the final speed of the PME's in RP3x, and the single-user mode in which it was normally used, the Mach/RP3 kernel was more system than was actually required on RP3x. We did not usually run RP3x as a time-sharing Unix system, so we did not really need a full-function Unix system. It may have been more appropriate to use a small run-time executive that implemented the small set of system calls that our applications actually used. The difficulty with this approach would have been choosing an appropriate set of system calls. New calls might have been required as more and more applications were ported to the machine. However, for operating system researchers, Mach was more interesting than a small run-time executive, and this factor played a key role in our choosing Mach.
- Mach is a large system. Changes to the system often required inspection of large portions of code that were irrelevant to the problem at hand. A small kernel might have been easier to deal with.
- Mach IPC on the Mach 2.0 kernel we ran was very expensive. Mach IPC was used not only for communication between user tasks, but also for communication between a user task and the kernel, at least for Mach-specific kernel functions. IPC performance has been improved in Mach 2.5 and subsequent releases, but for our kernel a Mach system call was significantly more expensive than a BSD system call, and the difference was largely due to IPC overhead.

6.3 Mach kernel serialization

We found it was not necessary to significantly restructure the Mach kernel to achieve significant application-level parallelism for computation intensive workloads. Mach/RP3 was a version of Mach 2.0. The true Mach portion of that kernel was parallel and symmetric, but the kernel included a lot of BSD Unix code that essentially ran under a single lock. That is, the Unix portion of the kernel was executed exclusively by a single processor called the *Unix Master*. *Slave* processors ran user programs, Mach code, and trivial Unix system calls. All other Unix system calls were implemented by suspending the calling thread on the slave processor and rescheduling the thread on the master processor.[4] Furthermore, because it was designed for more or less generic multiprocessors, the Mach kernel did not make significant use of the sophisticated RP3 *fetch&op* synchronization primitives.

The NYU Symunix system was designed specifically to avoid these limitations, but in our experience the problems were not severe. We were able to achieve 40-way speedups on the 64-way RP3x system with relative ease. We attribute this success to the fact that our workloads were usually engineering-scientific programs. A typical program issued a number of system calls to create separate processes and establish a common shared-memory region, but once it began its computation, it issued relatively few system calls. (To some extent the workload was artificial, since users knew that RP3x had limited I/O bandwidth and hence did not run I/O intensive jobs on the machine. Nonetheless, we feel it is a workload characteristic that such jobs issue far fewer system calls per million instructions than a commercial workload might.)

Originally, it was felt that more system restructuring would be necessary for RP3. For example, we expected we would have to modify the system dispatcher to use highly-parallel, non-blocking queue insertion and deletion routines based on *fetch&add* [10]. However, we never found the dispatcher on RP3x to be a significant bottleneck, in particular because our philosophy was to allocate processors to user tasks and to let users do local scheduling. The system dispatcher was

4. A group at Encore parallelized much of the Unix code in later versions of Mach [5]. These changes were subsequently picked up by the Open Software Foundation and are a part of the OSF/1 kernel.

only used for idle processors and global threads that were not bound to particular processors. The scheduling problem thus divided naturally into two levels: system-level scheduling decisions that were made on a job by job basis, and user-level decisions that were made on a thread by thread basis. (A two-level scheduling mechanism of this flavor is described in [4]). The intervals between system-level scheduler events were on the order of many seconds to a few hours; user-level scheduling events could occur as frequently as once every few hundred instructions. Thus the system-level scheduler was not a bottleneck and did not need to use *fetch&add* algorithms. The user-level scheduler was part of the application and if necessary could use a *fetch&add* queue in user space to reduce local scheduling overhead.

6.4 Architectural Compatibility

The RP3 software development effort benefited greatly from the processor and operating system compatibility between RP3 and the RT workstation, but there were pitfalls. On the plus side, this compatibility let us use essentially all the Mach/RT utility commands (*sh, ps, ed,* etc.) on RP3 without recompiling. We could concentrate on those areas of the system that had to be modified. We were also able to adapt and use the kernel debugger from Mach/RT as the kernel debugger for RP3.

We were very successful in using Mach/RT tools for program development on RP3. The compilers we used on a routine basis were the same ones we used on Mach/RT. (These compilers, in turn came from IBM's AOS 4.3 for the RT system.) Differences between Mach/RT and Mach/RP3 were encapsulated in library routines. (For example, a subroutine on the RT emulates the RP3 *fetch&add* operation using the RT *test&set* instruction.) Application programs compiled for the RT could be relinked to execute on RP3x. We were able to test user code on Mach/RT and then move it to RP3x for execution with relative ease.

Our efforts were both aided and complicated by the fact that a variety of floating point hardware options (including "no floating point hardware.") were available on the RT workstation. RT compilers did not generate native machine code for floating point hardware, but instead generated floating point pseudo-code. The pseudo-code was translated at runtime to native code appropriate for the floating point hardware actually found on the host machine. If the host machine had no floating point hardware, the pseudo-code was simply interpreted at runtime. This floating point emulation mode let us develop real RP3 applications long before floating point hardware was available on RP3x.

However, we quickly discovered that the floating point emulator maintained a simulated floating point register set at a fixed location in a process's address space, and that the emulation was therefore incorrect for applications with multiple threads sharing an address space. The problem existed on the RT, but it did not show up until we deliberately disabled the floating point hardware on our workstations. On RP3 machines, including the RP3 simulator, the problem showed up immediately. We had to change the emulation code to maintain perthread simulated register sets.

When floating point hardware was finally installed in RP3x, the runtime translation of floating point pseudo-code became a serial bottleneck. The translation was performed the first time a particular floating point operation was encountered during execution. Conversion involved replacing the pseudo-code sequence with compiled instructions appropriate for the floating point hardware. In a multithreaded application, the second and succeeding threads that hit a particular floating point operation had to wait while the first thread translated the pseudo-code. Furthermore, the compiler sometimes did not reserve enough data space to hold the translated code sequence, at which point the floating point code generator called *malloc* to dynamically allocate space for the generated instructions. The *malloc* could require a system call, seriously affecting application performance. This problem was particularly severe for non-threaded applications that forked lots of separate processes, because the multiple system calls from independent processes could interfere with each other in the kernel.

A final problem with runtime floating point code generation was that the generated code ended up in the read-write data segment of the program rather than in the read-only code segment. Parallel programs could not easily cache read-write data, so for the most part, parallel programs wound up fetching floating point instructions from memory rather than from cache.

Two incompatibilities between the RT and RP3 environments caused us some headaches. First, while Mach/RP3 shared most system software with Mach/RT, it shared filesystems with Mach/370.

The problem was that Mach/370, and therefore Mach/RP3, used a filesystem block size of 1024 bytes, while Mach/RT used a 512-byte block size. Certain utility programs, namely *ls* and *fsck,* were compiled for a particular block size. These programs had to be recompiled for RP3. The problem with ls was subtle. The program never actually failed, but sometimes it would not display some of the files in large directories.

Finally, the original design of the RP3 switch did not support the ROMP *partial-word-store* instructions (*store-byte* and *store-half-word*). If a processor encountered such an instruction, it would raise a program exception. The plan was to avoid all such instructions by using a new compiler. This proposed compiler never materialized. Instead we had to resort to post-processing the assembly language output of our C-compiler. In this manner we avoided partial-word-stores in the kernel itself and in application code that could be recompiled. To avoid having to recompile all user commands, we wrote a program exception handler to emulate the partial-word-store instructions encountered in user mode. This emulator had the following disadvantages:

- It could not be made to execute both efficiently and correctly on a multiprocessor. A lock was required to prevent simultaneous updates to different bytes of the same word.
- It was extremely slow.
- It required emulation of not only the partial-word-store instructions themselves, but also of those *delayed-branch* instructions that contained a partial-word-store instruction in the delay slot.

Using the RP3 simulator to demonstrate the software overhead of partial-word-store emulation, we convinced the hardware designers to change the switch design to support these instructions. In general, if compatibility with an existing system is a goal, it is extremely important to identify and match all the features of the existing system that are commonly used. It is a mistake to depend on a compiler or other sophisticated system software to gloss over inconsistencies.

6.5 Large-Scale Multiprocessors

We found that a 64-processor machine was much different from a 4-or 8-processor machine. A similar observation concerning the BBN Butterfly was made in [12]. We learned this lesson in October 1988 when we first tried to boot our kernel on the 64-processor machine. Before this, we had successfully run our kernel and a small set of applications on the 4-way and 8-way prototypes, and had booted the kernel under a 64-processor version of the RP3 functional simulator. On the 4-way and 8-way machines, it took only a few day's effort, once the hardware was available, to get our kernel and application suite running. (This success, of course, depended on much previous work with the simulator and with 1and 2-way versions of the hardware.) We did not encounter new timing bugs in the kernel when moving from the 2-processor to the 4- or 8-processor systems, and reasonable kernel startup times and application speedups were easily achieved. However, when we attempted to bring up the 64-way kernel, we found to our surprise that:

- New timing bugs appeared (the kernel would not boot reliably).
- Kernel startup time had expanded to an unacceptable 2.5 hours!

Of course, we knew that the kernel we were using at that time was far from optimal (it did not use interleaved memory, for example) but we were surprised nonetheless by the disparity between the kernel startup times for the 8-way and 64-way machines. Eventually we were able to reduce the kernel startup time to about 8 minutes by reducing network contention due to spin locks, by placing the kernel in interleaved memory, and by exploiting the processor caches on RP3x.

6.6 Processor Allocation

Traditional ideas of processor allocation and system control do not necessarily apply in the shared-memory parallel-processing arena. In [15], Pancake and Bergmark note the discrepancy between the approach to parallel programming taken by computer scientists and that taken by computational scientists. We encountered this distinction when we first started work on the RP3 operating system. We were somewhat shocked by the attitude of the (potential) RP3 user community toward operating systems and operating system scheduling. Instead of regarding the operating system as a convenient environment for parallel programs, some of our users regarded the system as an adversary bent on denying them direct access to the hardware.

For example, in our view the operating system had the right to suspend any process at any time based on the operating system's concept of the importance of that process. Our users explained to us, patiently, repeatedly, determinedly, and when necessary, vehemently, that this approach wreaked havoc with parallel programming models that did their own processor allocation. For example, in the EPEX model of computation, the program determined the number of processors available to the job and divided FORTRAN *DO*-loops across the available processors. Each processor was assigned a subset of the *DO*-loop indices for execution. Subsequently, a *BARRIER* statement was used to ensure that all loop instances were complete. Since it was assumed that the computation had been divided equally among the processors, the *BARRIER* was implemented using spin locks rather than suspend locks. Each processor was represented in the EPEX program as a separate Unix process. If the operating system were to suspend one of these processes after the parallel *DO*-loop has started, the remaining processors would loop endlessly when they reached the *BARRIER* statement, waiting for the last processor to complete its part of the computation.

Similarly, a job might only be able to adapt to a change in the number of processors at certain points in its execution. EPEX programs could not adjust to a change in the number of processors during execution of a parallel *DO*-loop. Only before the loop started or after it completed could the number of processors be changed. Even then, the allocation of additional processors required the creation of additional Unix address spaces in the EPEX application. The only realistic approach for the EPEX model was to statically allocate processors to the program when it began executing. This strategy, of course, conflicted with the ability of the system to run other jobs, to service interrupts, or to multiprogram the system, none of which were of interest to our users.

Since a significant number of applications for RP3 had already been written for the EPEX model of computation, we could not afford to simply ignore these concerns. Instead, we developed the concept of *family scheduling*[7] which corresponds to the ideas of *gang scheduling* or *co-scheduling*[14]. With the family scheduler extensions, the Mach/RP3 system would never suspend individual processes in a parallel program, but it could, if necessary, reclaim processors from a family by suspending the entire family.

We believe the family scheduler was an acceptable compromise between our users' needs and the requirements of the operating system to perform global scheduling. The EPEX run-time library was enhanced to use the family scheduling primitives on RP3 and the PTRAN [3] compiler, an automatic parallelizing compiler for FORTRAN, also used the family scheduling primitives. In the end, however, the facilities of the family scheduler were largely underutilized, because most users prefered to run applications on RP3x in a single-user-at-a-time mode in order to obtain repeatable execution times.

6.7 Performance Consistency

Obtaining consistent and repeatable performance measurements was a problem for us on RP3x. Our users were primarily interested in studying parallel algorithm speedup. This task was difficult if individual timing runs were not repeatable. Variations in execution time from one run to the next could be caused by a variety of factors.

On RP3, multiple jobs executing simultaneously could interfere with each other even if there were enough processing resources to satisfy all of their requirements. Contention for memory bandwidth, for processing time on the Unix master processor, and for I/O bandwidth could degrade the performance of even well-behaved programs. Most jobs were therefore run on the machine in a single-user-at-a-time mode.

Even on an otherwise idle machine, however, execution times could vary from one run to the next. Of course, the variance could be due to gross non-determinism in a true parallel program, but even subtle non-determinism could affect performance. In successive runs a given processor might execute exactly the same sequence of operations but on different regions of a shared data structure. Even if the data structure were laid out uniformly in interleaved memory, operations against different regions might result in different remote access patterns, and consequently in different levels of network and memory contention.

Variations in the execution times of completely deterministic applications could be due to non-deterministic placement of kernel data structures associated with the application. For example, process context blocks and per-process kernel stack segments were dynamically allocated when an application started. These structures might therefore reside at different kernel virtual addresses from one run to the next, and consequently might be spread across different sets of PME's. This problem was usually not severe for computation-intensive applications, although clock-interrupt overhead was occasionally increased because of a particularly unfortunate placement of a kernel data structure.

Our users learned to make sure the same version of the operating system kernel was used for an entire sequence of timing runs. Trivial changes to the Mach/RP3 kernel could cause timing differences in user applications, because the kernel itself was located in interleaved memory. Small changes in the kernel could shift key data structures in such a way that imbalances in memory reference patterns could appear (or disappear). Unfortunately, even simple bug fixes could change the layout of memory. In one case, a 50-byte change in initialization code (only executed at boot time!) dramatically changed key performance measures of a particular user program.

A final point concerns the RP3 address hashing mechanism. With hashing disabled, the PME location of a particular double-word of an interleaved page was determined solely by the word's virtual address. With hashing enabled, the location was a function of both the virtual and real addresses of the double-word. Applications could explicitly control the virtual addresses they used, but they had no control over (or even knowledge of) the real addresses they used. In repeated runs of a program, real memory addresses of the user's virtual pages would change, and hence remote access patterns would change from one run to the next, even if the application made deterministic use of its virtual address space.

6.8 Static Partitioning

We found that multistage interconnection networks do not lend themselves to construction of traditional multi-user machines. As mentioned above, we usually ran RP3x in a single-user-at-a-time mode. Given our application set and user community we would suggest that designers of other machines with memory hierarchies implemented by multi-stage interconnection networks not attempt to run the system with a single kernel, but instead adopt a partitioning approach similar to that proposed for TRAC [17].

On RP3, we could have used the variable interleave amount to statically partition an RP3 machine into sub-machines on power-of-two boundaries. To keep partitions from interfering with each other, each partition could have been given its own copy of the kernel, with each page in a partition being either a sequential page or a page interleaved across the entire partition. The machine would have been effectively split into distinct sub-machines as far as the switch and memory were concerned. Assuming that the I/O system were reconfigured to split along similar lines, this partitioning would have given users appropriately sized machines, with strong guarantees of repeatable performance and non-interference from other users.

Such a static partitioning scheme would have required a controlling system to run outside of RP3 itself. A natural place to run this system would have been the I/O and Support Processors or ISP's in the system since these machines already supported system start-up. We did not pursue this approach because the software to do so would have been complex and would have had to run in the primitive environment of the RP3x ISP's. However, this approach would probably have been a better match to both our users' needs and to the memory and processor structure of the machine. It would certainly have improved processor utilization over our single-user mode of operation, since a singleuser job that used only a few processors left most of the machine idle.

6.9 NUMA Programming

NUMA machines (like RP3) share the advantages and disadvantages of both tightly-coupled and distributed multiprocessors. On a message-passing machine, a serial program must usually be significantly restructured before it can even be loaded onto the parallel machine. The primary advantage of the shared-memory parallel processing approach is that it is relatively easy to get an existing, serial program to work in the parallel environment, and on RP3 we found this to be the case. Transformation of serial programs to the EPEX environment was relatively straightforward [8], and once an application ran under EPEX on the RT, it was a simple process to move it to RP3.

However, even though the program would run correctly on RP3 with relatively little restructuring, achieving the maximum available speedup usually required significant program restructuring. Multiprocessor speedup could be limited unless one used such RP3 features as local and cacheable memory.

For programs in the EPEX style, code and private data were replicated to each address space and thus could be placed in local and cacheable memory by the language run-time. With this optimization speedups in the 40's could be attained on the 64-processor machine with relative ease. To improve the speedup, additional code restructuring was required.

For example, one of our colleagues (Doug Kimelman) obtained a speedup of 57 on 64 processors for a parallel implementation of a prime number counting program based on the sieve technique. To achieve this speedup he had to:

- divide the sieve space up into 64 subranges.
- place each subrange in local memory on a processor.
- execute separate sieve algorithms on each processor.
- use *fetch&add* to accumulate the number of primes found in the subrange into the total number of primes found.

Exactly this kind of program repartitioning would also have allowed the sieve algorithm to work well on a message-passing machine.

The point is that in order to get good speedups on any kind of finegrain, highly-parallel computation, one must concentrate on partitioning the data in such a way as to minimize data movement and interprocessor synchronization. This statement is true for both shared-memory and message-passing architectures. On RP3, it is our belief that while it was easy to convert serial code to parallel code and to get it to execute correctly, obtaining maximum speedups required the same kind of restructuring that would be required to get the program to execute on a message-passing machine. We still feel that the overall effort was smaller on RP3 than on a message-passing machine, because the existence of shared memory let the programmer concentrate on restructuring only those parts of the program that were critical for good parallel performance. Distributed shared memory systems such as that of Kai Li [13] may alleviate this disadvantage of messagepassing systems, but the amount of restructuring required to achieve a given level of performance will still be greater on such systems than on true shared-memory systems because the penalties for non-local memory access are so much greater.

7. Large-Project Management

7.1 Design Restraint

The RP3 project as a whole suffered because the initial design was not focused tightly enough on the architectural features that made RP3 interesting. In our opinion, those features were the switch, the memory management unit, and the cache. The project invested a lot of effort in those features, but it also devoted a lot of time and money to the design and implementation of an experimental ISP (I/O and Support Processor) and of a floating point coprocessor based on a System/370 vector unit. Even if these efforts had been successful, the additional features would not have made the machine significantly more interesting, because it was really the memory subsystem that distinguished RP3 from other multiprocessors.

We are not saying the experimental ISP was not an interesting research project in its own right. The point is that it was experimental and therefore risky, and the overall project could not afford to take risks in areas unrelated to the truly novel aspects of the architecture. I/O is important, and when the experimental ISP was abandoned, we were left with an ad hoc I/O architecture that, in its "enhanced" version, achieved the unimpressive per-ISP bandwidth of about 500 kilobytes/second. This small bandwidth limited our ability to experiment with performance visualization [11] , to study seismic algorithms, and to run graphics applications on RP3, and it limited the speed of routine interaction with RP3x. The project would have been more successful if part of the effort that went into the original ISP had instead been devoted to the implementation of a reasonable, if unspectacular, interface to a standard I/O subsystem.

The floating point story is similar. Instead of borrowing one of the floating point hardware implementations from the RT workstation, the initial RP3 PME design incorporated a vector floating point unit from a midrange System/370. The design was experimental, because no one had before tried to marry a ROMP processor to a System/370 floating point engine. Furthermore, use of the vector unit presupposed the completion of a new compiler capable of generating appropriate floating point code. (The same compiler was supposed to solve the partial-word-store problem.) The effort required to complete the vector hardware and the supporting compiler was seriously underestimated, and neither was ever completed. As a result, RP3x was without floating point hardware of any kind for a year and a half, a factor that contributed greatly to the disillusionment of our potential user community.

A late effort to retrofit an RT-compatible floating point coprocessor into the initial PME design was straightforward and successful, and it increased the speed of typical compiled floating point code by a factor of 40, compared with software floating point emulation. As with I/O, the RP3 project would have been more successful if this strategy had been pursued in the original design.

7.2 Future of Shared-Memory Multiprocessing

The success of large-scale shared-memory multiprocessors has been limited because it is hard to stay ahead of the RISC microprocessor development curve using parallel processing. Since we started the RP3 project in 1984, we have seen the development of office workstations with 15 times the processing power of the original RT. Advances in RISC processor technology continue, with processor speeds doubling every year or two. If in this environment a parallel processing machine is built based on a particular microprocessor, and if the lead time for this development effort exceeds a few years, the resulting machine will be regarded as obsolete when it is completed, unless, of course, the then current-generation processor can replace the original processor designed into the machine. Thus, in order for large numbers of microprocessor MIPS to be effectively harnessed into a large machine, it is crucial that the additional complexity associated with the machine interconnection hardware be kept as small as possible, or that the machine be built in a sufficiently modular way that it can accept upgraded processor chips.

Machines like RP3, which were designed to the characteristics of a particular microprocessor, and which required large investments in chip design, manufacturing, and packaging, have long development cycles and are ill-suited to compete against the increasing speed of commodity microprocessors. It is simply too hard to stay ahead of the RISC processor development curve. The memory interface of the microprocessor may not be sufficiently well architected to allow one to move to nextgeneration chips in a shared-memory parallel processor.

On RP3x, we were limited to a slow cycle time due to timing errors in the network interface chip, but even if we had not been so limited, we could not have switched to the faster CMOS ROMP without a redesign of the RP3x processor card.

Message-passing machines, on the other hand, have a much simpler and more modular interface between the processor and the interconnection network, and so are easier (and quicker) to build, and easier to upgrade when nextgeneration chips become available. For massively parallel applications and environments where the user is willing to completely rewrite programs to achieve a performance improvement of several orders of magnitude, these machines should be the vehicle of choice for parallel processing. For programs too large to be restructured easily, and for automatic parallelizing compilers, the future remains in shared-memory parallel processing with a modest number of processors.

7.3 Overstated Goals

The RP3 project is widely perceived to have been unsuccessful. In part, this perception arose because of the disparity between the performance of the RP3x prototype and performance predictions published in the 1985 ICPP proceedings [1,6]. In [1], RP3 was predicted to achieve a peak performance of 1.2 BIPS (assuming a 100% cache hit rate on instructions and data). Needless to say, RP3x did not achieve that mark, but it missed by only a factor of 3 when the prediction is scaled for a 64-processor prototype of the 512-way architecture: 1.2 BIPS per 512 processors is 2.3 MIPS per processor. In user mode, with data and instructions cacheable we routinely executed at 750 KIPS. The remaining factor of 3 is explained by the fact that we ran at one-third the clock rate originally proposed. This decrease in clock rate was made to circumvent timing errors in the final revision of the network interface chips. Rather than wait for yet another chip revision, the project decided to push forward and complete a somewhat slower machine. This decision was reasonable and defensible for a research project. Why is it then, that RP3 is regarded by many people as an uninteresting machine?

We feel this is due to several factors:

- When RP3x was first assembled in 1988 it included neither processor cache nor floating point hardware. When a system designed to include cache and hardware floating point is run without cache and with software-emulated floating point, performance is bound to be disappointing. The final upgrade of the machine to include an RT-equivalent floating point unit was not completed until 1990, and in the meanwhile a number of our potential users developed a negative impression of the project as a whole.

- Since the NMOS ROMP in RP3x was announced by IBM in the original RT, a new version of the RT (the APC, a CMOS ROMP) was delivered, and it in turn has been replaced by a new generation RISC workstation, the RISC System/6000. The floating point performance of a single RISC System/6000 exceeds the performance of the 64-processor RP3x. But the point of RP3x was to enable research into parallel processing, not to compete with the fastest processors available. RP3x was always slower than a CRAY Y-MP, for example.
- Flexible, but slow hardware is not as interesting to users as inflexible, but very fast hardware. In the end, the success of RP3x was determined by its users. We wanted to build a machine that was flexible and sufficiently fast to attract application users to the machine. The hardware was very flexible, but unfortunately, it was not fast enough to attract applications. Instead, only the parallel processing expert was interested in using the machine, while other users moved to faster hardware.

The real problem we feel is that the goals of the RP3 project were overstated. We met the realistic goal of creating a reliable research machine oriented to studying the hardware and software aspects of parallel, shared-memory computation. We believe the project would be regarded more favorably today if we had not initially claimed to be building the world's fastest computer.

8. Concluding Remarks

The RP3 project was a large and ambitious project whose goal was to build a flexible research vehicle for studying hardware and software aspects of shared-memory parallel processing. RP3 was never intended to be a product, and hence it should not be judged in comparison with product machines, but as a research vehicle. In that light we would argue that the project was successful:

- The machine was completed and was a reliable research tool for more than two years.
- We were able to develop a version of Mach for RP3 that exploits the memory, cache, and processor architecture of the machine and lets programmers use these facilities to create efficient parallel programs on RP3x.
- We demonstrated that 40-way speedups are relatively easy to achieve on a 64-way parallel processor, and we learned much about the use of such a machine for parallel processing, as judged by the experience reported in this paper.

We hope that some of the lessons we learned can be of help to others building similar machines, whether they are intended as research prototypes or product machines.

9. Acknowledgements

Our work on RP3 could never have been completed without the assistance of many people, without whose conscientious efforts the system would not exist. In particular, we would like to thank Fran Allen, Gordon Braudaway, Wally Kleinfelder, Matt Thoennes, and Herb Liberman for their assistance and support during the RP3 project. Rajat Datta was responsible for the RP3 I/O service machines on VM; Rajat also contributed significantly to the first port of Mach to an RP3 PME. Tony Bolmarcich was responsible for the implementation of EPEX for Mach/RT and Mach/RP3. The original port of Mach to the RP3 simulator was completed by Dan Julin of Carnegie Mellon University while he was a summer student working with 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. Dan Rencewicz of IBM/TCS at CMU was responsible for obtaining production releases of Mach for the RT system and making them available within IBM.

References

[1] M. Accetta et al., "Mach: A New Kermel Foundation for Unix Development," *Usenix Association Proceedings,* Summer 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *The Journal of Parallel and Distributed Computing,* vol. 5, no. 5, pp. 617-640, Oct 1988.

[3] T. E. Anderson, B. Bershad, E. Lazowska, and H. Levy, Scheduler Activation: Effective Kernel Support for the User Level Management of Parallelism, Department of Computer Science, University of Washington, October 1990. Technical Report #900402.

[4] J. Boykin and A. Langerman, "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis," *Usenix Workshop on Experiences with Distributed and Multiprocessor Systems,* pp. 105-126, Fort Lauderdale, Florida, 1989.

[5] W. C. Brantley, K. P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference on Parallel Processing,* pp. 782-789, August 1985.

[6] R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg, "Operating System Support for Parallel Programming on RP3," *IBM Journal of Research and Development,* Submitted for publication.

[7] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/Fortran," *Parallel Computing,* no. 7, pp. 11-24, 1988.

[8] J. Edler, J. Lipkus, and E. Schonberg, Process Management for Highly Parallel Unix Systems, New York University, 1988. Ultracomputer Note #136.

[9] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Coordinating Large Numbers of Processors," *ACM TOPLAS,* January 1982.

[10] D. N. Kimelman, "Environments for Visualization of Program Execution," *IBM Journal of Research and Development,* Submitted for publication.

[11] T. J. LeBlanc, M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN PPEALS 1988—Parallel Programming: Experience with Applications, Languages, and Systems,* pp. 161-172, 1988.

[12] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proceedings of the 5th Symposium on Principles of Distributed Computing,* pp. 229-239, August 1986.

[13] J. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. of Distributed Computing Systems Conference,* pp. 22-30, 1982.

[14] C. M. Pancake and D. Bergmark, "Do Parallel Languages Respond to the Needs of Scientific Programmers?," *IEEE computer,* pp. 13-23, December 1990.

[15] G. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing,* pp. 764-771, August 1985.

[16] G. F. Pfister and V. A. Norton, " 'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *Proceedings of the 1985 International Conference on Parallel Processing,* pp. 790-795, August 1985.

[17] M. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *Proceedings of the 1980 National Computer Conference,* pp. 631-641, 1980.

[18] R. H. Thomas, "Behavior of the Butterfly Parallel Processor in the Presence of Memory Hot Spots," *Proceedings of the 1986 International Conference on Parallel Processing,* pp. 46-50, August 1986.