Mach/4.3BSD: A Conservative Approach To Parallelization

Joseph Boykin and Alan Langerman
Encore Computer Corporation

ABSTRACT: Mach is a new operating system targeted for distributed and multiprocessor environments. Mach contains 4.3BSD compatibility code that, unlike the Mach kernel proper, runs only on a single processor, thus presenting a performance bottleneck to a multiprocessor system. Pieces of the 4.3BSD compatibility code were selectively parallelized to reduce this bottleneck. Significantly improved multiprocessor and multi-user performance was achieved using minimum modification of existing data structures and algorithms. A framework was left in place for future parallelization enhancements.

This research was supported in part by the Defense Advanced Research Projects Agency (DoD) through ARPA Order No. 5875, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1. Introduction

The Mach operating system, developed at Carnegie-Mellon University, targets a broad range of computer architectures, including uniprocessor, multiprocessor and distributed systems. The designers of Mach intend to produce a compact, efficient kernel on top of which may be layered interfaces for traditional operating systems such as 4.3BSD, System V, MS-DOS, VMS, etc. Most traditional kernel support, such as device drivers and filesystem handling, will be provided by a set of user-level servers. The Mach kernel will provide the mechanisms necessary for simple operation in a distributed environment using uniprocessor or multiprocessor systems. Mach currently provides full backward compatibility with 4.3BSD. However, while Mach exploits the full power of a multiprocessor the 4.3BSD compatibility code does not; we have parallelized large portions of this compatibility code while retaining the original data structures and algorithms. The result has been a kernel that yields good multiprocessor performance.

Encore is interested in Mach because of its multiprocessor support [Boykin & Langerman 1989; Langerman et al. 1990]. In particular, DARPA sponsors Encore's development of a 1,000 MIPS multiprocessor that will use Mach. Encore currently runs Mach on the Multimax, a symmetric shared memory multiprocessor using the National Semiconductor 32000 family of processors.

Mach uses original 4.3BSD code to insure BSD compatibility. As currently distributed by CMU, Mach's 4.3BSD compatibility code has not been modified to support efficient multiprocessor operation. The original 4.3BSD kernel was designed for a uniprocessor: kernel data structures are protected from interrupt-level

race conditions by disabling interrupts at appropriate times. This approach does not suffice in a multiprocessor environment in which processors may be using shared data structures simultaneously and interrupts may be processed on any available processor.

The Mach kernel is designed and implemented to execute correctly on a multiprocessor. Mach uses multiprocessor locks to synchronize operations between separate processors. These locks include spin locks (called *simplelock*s) for non-blocking synchronization and read/write locks that may cause a thread to suspend until the lock becomes available. Mutual exclusion locks are built from read/write locks. *simplelock*s may also be used to synchronize between processors and I/O devices that operate out of main memory.

Mach resolves the contradiction between the native, inherently parallelized Mach code and the inherently serial 4.3BSD compatibility code by forcing all 4.3BSD code to execute on a single processor, the so-called *master*. We use the term *unix_master* to denote this restriction because the internal Mach function *unix_master*() forces a Mach thread to execute on the *master* processor. Device interrupt handling is also confined to the *master* processor. Thus, the normal 4.3BSD mutual exclusion mechanisms continue to operate as expected. Obviously, any Mach code that manipulates 4.3BSD state must also be restricted to the *master* processor.

The *master* processor design works well: all user-level code and all native Mach operations (e.g., Mach kernel calls, virtual memory handling and Mach IPC) execute on any available CPU. Only 4.3BSD-specific routines and the Mach code that interfaces directly to them must obey the *master* processor restriction. Ultimately the 4.3BSD compatibility code will migrate into user-level servers and become executable by any processor.

In the meantime, unfortunately, the *master* processor restriction has severe implications for overall multiprocessor performance. We observed that apparent Mach performance was significantly worse than that offered by the other Encore operating systems, UMAX4.3 (based on 4.3BSD) and UMAXV (based on System V). Even though the basic Mach functionality had been written from scratch for multiprocessor operation, the vast bulk of user code makes heavy use of the 4.3BSD compatibility code. It

became clear that the 4.3BSD routines had to be modified to provide better performance.

We realized that the *unix_master* restriction offered us the opportunity to parallelize the 4.3BSD compatibility code selectively. Rather than alter all of the 4.3BSD code simultaneously, we could modify one piece at a time for multiprocessor operation and examine the results.

We adopted these goals:

1. Minimize modifications to existing code.
2. Provide a framework for future performance enhancements.
3. Achieve significant performance increase with minimum work.

We sought to maximize multiprocessor performance with the least effort. In effect, we followed a "90/10" rule: try to capture 90% of the possible performance improvement at a cost of 10% of the total work. (We didn't take this maxim literally, of course.) Because of our resource limitations, we preferred to implement a framework for future parallelization and tuning efforts rather than parallelize all subsystems immediately or implement highly parallel subsystems from scratch.

After analyzing system call counts and interrupt handling, it became clear that the greatest performance wins were to be found by parallelizing the low-level interrupt handling, the filesystem, tty, and network code. In general, we parallelized code by adding synchronization mechanisms to existing data structures and adding appropriate calls to synchronization routines from existing algorithms. In other words, minimum modification was a cardinal rule.

The minimum modification rule was also important because we track functional modifications and bug-fixes to this code by Berkeley, CMU, and other organizations.

While a significant amount of work has already been done in the area of multiprocessor UNIX operating systems [Bach & Buroff 1984; Barton & Wagner 1988; Hamilton & Code 1988; Sinkewicz 1988], we are unaware of any design that incorporates an incremental approach to parallelization and attempts to achieve substantial parallelism without altering data structures or devising new algorithms from scratch. There is certainly no other

implementation that must reconcile these goals within the context of an operating system that is highly parallel in some parts but uses a master/slave relationship for the rest of the code [Rashid 1986].

We will describe some of the design decisions we made and implementation problems we encountered during the parallelization effort. First, we will focus on converting interrupt-level synchronization problems into multiprocessor synchronization problems. Next, we will discuss our modifications to the 4.3BSD filesystem and network code. We will also discuss our approach to debugging and statistics gathering. Finally, we will summarize our results and mention possibilities for future work.

We assume that the reader is familiar with the internals of the 4.3BSD kernel, particularly the filesystem and network code. The reader should also be aware that Mach uses tasks and threads, not UNIX processes, and throughout this paper we will use the Mach terminology. The original Encore Mach port, with no modification of the 4.3BSD compatibility code, was known as Encore Mach/0.2 and derived from CMU's Release 2.0 of Mach. The current release of Encore's Mach, including the parallelized 4.3BSD code, is known as Mach/0.5.

2. *Interrupt Handling*

A consequence of the Mach *unix_master* design is the restriction of all interrupt handling to the *master* processor. The same processor that executes the 4.3BSD code must also execute the interrupt handling code or the 4.3BSD programming model will break. This I/O restriction is doubly ironic in our symmetric multiprocessor as other processors capable of handling the interrupts go idle while the load on the *master* processor increases.

The parallelization of the filesystem, tty, and network further demanded that interrupt handling be "fixed" because the 4.3BSD-style interrupt handling would not function with a system using blocking locks. Left untouched, interrupt-level operations could attempt to take blocking locks with disastrous results.

We defined three somewhat conflicting goals for upgrading the 4.3BSD interrupt model for our multiprocessor environment:

1. Minimize work done at interrupt-level.
2. Transform interrupt-level synchronization problems into thread context synchronization problems (so multiprocessor locks could be used).
3. Avoid lengthy processing delays, where possible.

We chose to define new kernel threads that would be responsible for handling incoming interrupts. The interrupt handler would be responsible for saving appropriate information and then waking up the appropriate thread to complete the processing. For example, the Multimax has four main interrupt sources: per-processor time-slice end counters; the System Control Card (SCC) which, among other things, provides serial ports for local and remote consoles; the masstore (disk/tape) interface; and the Ethernet interface. Time-slice end activities are already handled by the Mach kernel and therefore required no additional work on our part.

2.1 Console TTY handling

The interrupt handler for the directly-connected serial ports required some recoding. Originally, the SCC interrupt handler, *slcintr*, would directly invoke SCC tty routines. In our parallelized code, however, the SCC tty routines must acquire a blocking *tty_lock* before manipulating tty data structures. We modified *slcintr* to catch the interrupt, enqueue a unit identifier on the *scc_pend_intrs* queue, then awaken the *slcintr_thread*. The *slcintr_thread* handles the normal character processing, including calling into the SCC tty routines. Keeping up with console input is not difficult and we don't mind a delay between receiving the character and processing it so the *slcintr_thread* has a relatively low priority.

2.2 Masstore Interrupts

We have paid more attention to optimizing the handling of masstore interrupts because they are frequent and important. A masstore interrupt signals the completion of an I/O command or the generation of an error message. *msintr*, the masstore interrupt

handler, reads, logs and discards error messages. This behavior need not change for parallelized interrupt handling. However, on an I/O completion, there may be a need to manipulate the buffer on which the I/O finished. The non-parallelized *msintr* always called into a buffer cache routine, *iodone*, to pass on news of the I/O completion. *iodone* might then call *brelse* to release the buffer back to the buffer cache. All of these activities took place at interrupt-level. In our parallelized filesystem, however, blocking locks synchronize access in the buffer cache. It is an error for the interrupt-level code to manipulate blocking locks.

We created the *biodone_thread* to process all I/O completions. *msintr* queues information about the I/O completion to the *biodone_thread*, which wakes up and calls *iodone*. Blocking locks can then be acquired in thread context.

However, the *biodone_thread* itself can become a bottleneck in the disk subsystem; typically, there is only one thread and there is also a rescheduling delay when the thread is awakened. Furthermore, the thread will be used frequently, stealing time from other running threads. To alleviate these problems, we optimized the frequent case of a synchronous I/O completion to avoid using a *biodone_thread* at all. Normally, for a synchronous I/O, *iodone* merely has to wake up the user thread waiting for the I/O to complete; no buffer cache manipulation is needed. Therefore, we employed an "event" mechanism that allows us to post the news of a synchronous I/O completion directly from interrupt-level, awakening the sleeping thread without using the *biodone_thread* or *iodone*. (Asynchronous completions, which manipulate buffer cache state, continue to require the *biodone_thread* and *iodone*.) This optimization substantially reduces the need for the *biodone_thread*. The design and implementation permit multiple *biodone_thread*s to be started in case a single *biodone_thread* becomes a bottleneck. Statistics to date suggest that a single *biodone_thread* is adequate.

2.3 Ethernet Interrupts

Interrupts from the Ethernet interface result from incoming packets, completions for outgoing packets, and error conditions. The latter two conditions are easy to handle and were already correctly

implemented for multiprocessor operation. The most important matter is handling incoming packets.

It should be no surprise that the original code would not work in a multiprocessor environment. The original algorithms would process packets and massage protocol information from the network interface all the way up to the socket layer while operating the whole time at interrupt level. This design was changed to minimize the work done at interrupt-level and because operations at interrupt-level can not work with blocking locks.

There are three parts to the solution. As in the original code, when the packet arrives, the interrupt handler determines the packet types and selects a destination queue for the packet (e.g., *ipintrq*). These queues are instances of *ifq*s, manipulated by a well-defined set of macros. We modified these macros (*IF_ENQUEUE*(), *IF_DEQUEUE*(), etc.) to operate in a multiprocessor environment using spin locks so that the macros could be used without change at interrupt-level and in thread context.

Having queued the packet, we awaken a `netisr_thread`. The `netisr_thread` invokes the appropriate protocol's incoming packet processing routine (e.g., *ipintr*) and normal packet processing continues except that the packet is now handled in thread context rather than at interrupt-level. Multiple *netisr_thread*s permit parallel processing of incoming packets; the number of *netisr_thread*s is configurable.

The last problem was to ensure that the queues to the intelligent Ethernet controller (the EMC) were locked to keep the queues consistent when multiple threads attempted to enqueue and dequeue packets. This was accomplished with a spin lock as these queues are also manipulated at interrupt-level.

For historical reasons, a separate thread was invented to handle incoming ARP requests. This thread could be eliminated today but there is no strong reason to do so. ARP traffic is relatively rare.

There were a number of other, lesser problems with interrupt handling that we do not have space to recount. The problems mentioned above were the most interesting and the most representative.

3. Filesystem Parallelization

The 4.3BSD filesystem code distributed with Mach is essentially identical to the filesystem code distributed by Berkeley. Some small modifications have been made at CMU but the scope of those changes is small and therefore irrelevant to our discussion. The following discussion applies to generic 4.3BSD-based filesystems.

3.1 Design Rules

Wherever possible, we exploited "natural" data structure parallelism. It was clear that the filesystem offered significant opportunities for data structure parallelism: *a priori*, there was every reason to believe operations could proceed in parallel on separate disks, filesystems, file descriptors, file structures, inodes, buffers, etc. It was also clear that operations could proceed in parallel against separate elements within important tables, like the inode and buffer cache hash chains. Most importantly, the natural structuring of the filesystem code implied that there were few potential deadlock problems between locks held at the various filesystem layers. For example, a thread could acquire (in order) a file structure lock, an inode lock, a buffer lock and device driver locks without deadlocking with other threads performing similar activities. On the other hand, there were some interesting races within the various layers. There were small but easily resolved problems with interrupt-level code (see Section 2).

We did not need to re-design any of the existing 4.3BSD filesystem data structures, even where those data structures were internal and had no on-disk representation.

Initially we used only blocking, mutual exclusion locks to simplify implementation and ease debugging. As the code matured we migrated to read/write and *simplelock*s.

In the Encore Mach/0.5 release, most filesystem code has been parallelized, including the tty subsystem and all interrupt-handling code. There are a number of subsystems that remain unparallelized. The various CMU-developed remote filesystems, RFS and VICE, have been modified to work in conjunction with the

parallelized filesystem code, chiefly by taking and releasing filesystem locks at the appropriate times. This is not to say that these subsystems have been parallelized; they still depend on the *unix_master* restriction because the RFS- and VICE-specific code and data structures have not themselves been parallelized. Other major subsystems that have not been treated include quotas and a CMU-specific pseudo-tty implementation.

3.2 Implementation Details

The scope of the filesystem parallelization effort is too broad to recount in detail. Instead, we will discuss some of the interesting cases encountered in the implementation.

The most challenging subsystem to parallelize turned out to be the buffer cache. The relationships among the hash table, the various freelists, and the buffers themselves are complex and further complicated by the different ways the cache can be accessed from interrupt-level and from within thread context. Interrupt-level buffer cache manipulations had to be eliminated, as we described in Section 2.2.

The internal complexity of the buffer cache led to a large number of possible deadlocks. Most of these deadlocks were resolved without restructuring the underlying algorithms by using conditional locking. With conditional locking, a thread receives an error indication if acquiring a lock would require blocking. For example, when fetching a disk block from the cache, it is necessary to lock the hash chain where the buffer containing the block should go, search the chain and, on a miss, allocate an empty buffer from the free list. However, buffers on the free list are also linked onto hash chains and must be removed from those chains. Naively acquiring the second hash chain lock could deadlock. Releasing the first hash chain lock opens up new races and at a minimum requires re-locking and re-searching the hash chain after a buffer has been allocated from the free list. We chose to attempt a conditional lock on the second hash chain and, if the lock attempt failed, to try allocating a different buffer from the free list.

The buffer cache returns locked buffers to callers, so that the calling code does not have to be modified to understand buffer

locking. A substantial amount of code did *not* have to be altered because of this implicit locking. For example, cylinder group information is fetched through the buffer cache and operated on within the buffer itself. The buffer lock implicitly protects the cylinder group data, permitting significantly easier parallelization of the disk block allocation and de-allocation code.

That same disk block allocation code provides a good example of the use of our parallelization framework. At an early stage in the filesystem parallelization process, all of the disk block allocation code was single-threaded through a disk block allocation lock (*disk_alloc_lock*). This scheme allowed us to bring up the filesystem quickly as only the few routines used outside of the disk block allocation package (e.g., *bmap*, *ialloc*, *ifree*, and *dirpref*) had to be modified to take the *disk_alloc_lock*. There were no race conditions to consider and the implementation took very little time. Once we had the filesystem running and had achieved basic stability we analyzed lock contention and found it to be unacceptable. The solution was to migrate to a scheme using the implicit cylinder group locks described above. However, it was also necessary to lock accesses to the in-core superblock at appropriate times and guarantee that there were no deadlocks between superblock locks, (implicit) cylinder group locks and other filesystem locks.

At a higher level, we encountered a number of interesting problems with file descriptors and file structures. Mach permits all of the threads in a task to share the task's file descriptor table. It is then possible for one thread in a task to be altering the descriptor table while another thread is using it. We defined individual locks for each file descriptor to allow as much parallelism through this table as possible. We envisioned utilities like parallel make, find, and grep that would be heavy file descriptor table users. The individual locks created their own problems: for example, two threads within the same task trying to *dup2*(2) could deadlock trivially if the first thread attempted a *dup2*(X,Y) while the second thread attempted a *dup2*(Y,X). For any situation requiring the acquisition of two file descriptor locks, we ordered the lock attempts by lock address to guarantee that no deadlock could result.

The interactions between pathname to inode translation (*namei*), inode fetching (*iget*) and filesystem attaching and

detaching (*smount*, *umount*) become slightly more complex in a multiprocessor environment. *iget* must cross mount points from the top of the filesystem hierarchy on down; *iget* detects mounted-on inodes and automatically fetches the root inode of the mounted filesystem. *namei* performs the opposite task: when translating ". ." in pathnames it occasionally must cross a mount-point going back up the filesystem tree.

In both cases, the original code "knew" that a filesystem could not be added to or removed from the mount table while *namei* or *iget* was active. In our multiprocessor kernel that assumption becomes invalid. The mount table was given a read/write lock, providing maximum parallelism for frequent operations, *viz.*, *namei* and *iget*, and adding minimal complexity to *smount* and *umount*. Had we used a mutual exclusion lock, *namei* and *iget* would have serialized across mount-points. On the other hand, a flag-based mechanism or some other lock that couldn't be held across an I/O would have significantly complicated the *smount* and *umount* code. By taking the *mount_table_lock* for writing, the *umount* code prevents *namei* and *iget* from crossing mount-points, thus making it easy to determine whether a filesystem is inactive. *smount* holds the *mount_table_lock* write-locked to eliminate other races. Since *smount* and *umount* are both infrequent operations, the typical case where the *mount_table_lock* is held read-locked presents no bottleneck whatsoever.

There were a number of minor annoyances related to the use of global variables. One embarrassing instance occurred with the *bmap* subroutine. We overlooked the read-ahead variables, *rablock* and *rasize*, maintained so that the callers of *bmap* know what block to request on a read-ahead operation. This omission on our part turned out to be insidious: for a very long time we weren't aware that there was any problem at all. The read-ahead variables were frequently over-written by another thread before they could be used by the thread that originally set their values. The resulting buffer read-ahead calls were nearly useless. Because the failure resulted in decreased performance but not in system failure (panic) we had no reason to suspect the existence of the problem. In fact, the problem was finally detected only because we noticed an unusual number of read-ahead calls into the buffer cache for disk blocks that should not have been the target of

read-ahead operations. We eliminated the global variables and forced *bmap* users to supply call-by-reference read-ahead variables.

Encore Mach/0.5 eliminated the *unix_master* restriction for roughly four dozen frequently used filesystem calls. In fact, only a few of these calls are heavily used but parallelizing those required modifying data structures used by the others. We were thus rewarded with a large number of parallelized filesystem calls "for free."

3.3 Performance Analysis

3.3.1 The Benchmark

The performance analysis effort used the Neal Nelson Business Benchmark [NNB 1986], a commercially-available set of system benchmarks. The NNB is oriented towards traditional UNIX filesystem operations. While Mach has a notion of memory-mapped files (and this notion has become popular in various UNIX dialects) we were more interested in characterizing the improvements we had made to the 4.3BSD compatibility code. The NNB fit the bill: it is simple to use, popular, and results are available for a wide variety of systems.[1]

The Neal Nelson Benchmarks consist of 18 separate tests oriented towards measuring filesystem and processor performance. Space limitations force us to confine our discussion to only four of those tests. Here are brief descriptions of them:

Test #1. "The Average User": various calculations and filesystem functions intended to represent the average user at work.

Test #3. Disk I/O: 250 iterations of a loop with a mixture of filesystem I/O functions.

Test #8. 500K Function Overhead Loop: call an empty function many times.

Test #18. Random Disk Tests: random reads from the disk.

1. The results we obtained are used only for comparisons internal to Encore. The data derived from the NNB suite are reprinted here in the format required by, and with the permission of, Neal Nelson and Associates.

The NNB driver is compiled with an option to select the maximum number of users to simulate during the benchmark run, typically between 20 and 60. During the course of the run, the driver executes a test program with arguments that select one of the 18 tests. The driver begins by executing one copy of the test program and recording the completion time for the test. The driver then executes two copies of the test program, as nearly simultaneously as it can manage, and records the completion times for those tests. This process is repeated until the driver has executed up to the maximum number of test copies requested.

3.3.2 Test Conditions

The NNB suite was run on a Multimax-320 configured as follows:

- 3 APC-01 CPU boards, 2 two-MIPS NS32332 CPUs per card, total 12 MIPS
- 2 SMC-16 memory cards, at 16 megabytes each, total 32 megabytes
- 1 EMC-I, with one Ethernet interface and one masstore interface
- 1 CDC Sabre 1.2 gigabyte disk drive, with average access time of 8.3 ms.
- 1 SCC, the System Control Card (irrelevant to this discussion)

As with all NNB runs, the system was brought to multi-user mode and a representative of Neal Nelson Associates downloaded and executed the benchmark. There were no other users logged in. There was substantial overall network traffic but only broadcast packets were sent to the benchmark machine. Network packets were therefore processed by the system; however, we presume that all benchmark runs should have been affected to approximately the same extent. We also ran unofficial benchmarks from single-user mode with the network interface disabled and achieved nearly-identical results; the differences were statistically insignificant. A single *biodone_thread* was present and active as needed. The *slcintr_thread* was present and would have been active whenever the console presented input to the system so the console was not used.

Both Mach/0.2 and Mach/0.5 booted from the same root partition and shared the same user partition. The NNB suite resided on the user partition and all working files for the suite were contained on that partition, as well.

The NNB was compiled for 20 users. (At larger numbers of users, the tests take a long time to run. In the future, we hope to have the opportunity to reserve a test machine for sufficient time to run a 60 user test.) The entire suite was run against Mach/0.2, the "serial" kernel, and Mach/0.5, the "parallel" kernel.

3.3.3 Test Results

The overall results indicate that Mach/0.5 does a substantially better job of exploiting the parallel architecture of the Multimax than does Mach/0.2. We will discuss some specific cases first and close with the most general test. The compute-bound tests, such as NNB #8 (see Figure 1), revealed no significant performance improvement in Mach/0.5 over Mach/0.2. Although the graph shows a small difference between Mach/0.5 and Mach/0.2, the difference is largely attributable to round-off error. All of the tests are coded to record only the time consumed by their CPU-bound

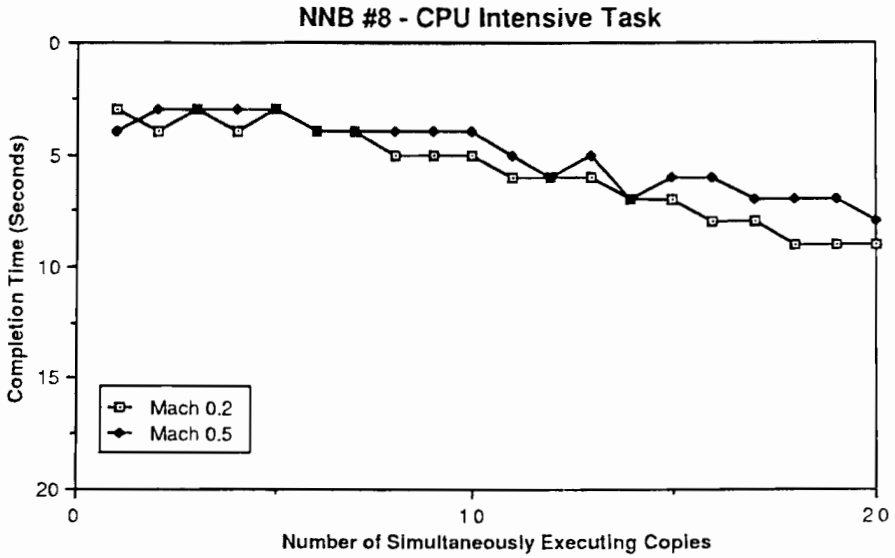

Figure 1: CPU-Bound Jobs under Mach/0.2 and Mach/0.5

portions, and both Mach/0.2 and Mach/0.5 distribute user-level computation to any available processor, so both versions of the operating system delivered similar results on the compute-bound benchmarks. This test is included as a control. NNB #18 yields more relevant results (see Figure 2). This test *lseek*s and *read*s from different parts of a working file. Each simultaneously executing copy of the test has its own working file. The test demonstrates a significant performance improvement for approximately 6-10 simultaneously executing copies of the test. However, Mach/0.2 degrades more slowly than we would expect and at roughly eight simultaneous tasks Mach/0.5 degrades surprisingly quickly, approximating the performance of Mach/0.2 from eleven through twenty simultaneous tasks. The primary culprit appears to be the *bfreelist_lock*, which our statistics demonstrated to have a miss ratio an order of magnitude worse than the next most frequently used lock. The *bfreelist_lock* is occasionally held for long periods of time while walking the buffer freelist or while waiting on a buffer lock. NNB #3 tests disk I/O by explicitly seeking to the beginning of the working file and performing five sequential 512-byte reads followed by five sequential 512-byte writes, after which random seeks and reads are done against the working file. This

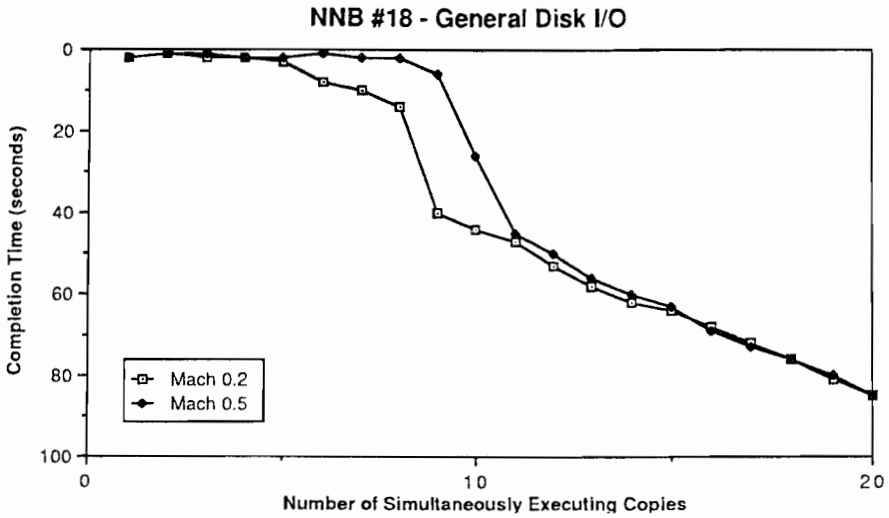

Figure 2: Random Disk Tests under Mach/0.2 and Mach/0.5

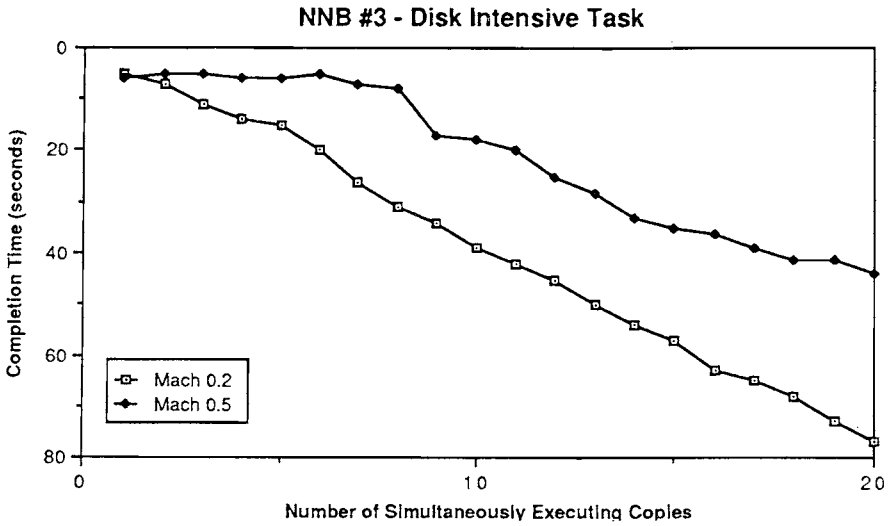

Figure 3: More Disk I/O on Mach/0.2 and Mach/0.5

loop is repeated 250 times. Once again, each task has its own working file. Mach/0.5 clearly out-performs Mach/0.2 until about eight simultaneous tasks, when decay sets in (see Figure 3). The main factor once again appears to be the *bfreelist_lock*, which displayed an unusually high miss ratio on this test as it did on test #18. NNB #1, representing the average user at work, nicely summarizes the current level of filesystem parallelization (see Figure 4). While the Neal Nelson Benchmark suite suggests that Mach/0.5 suffers from one or more as-yet-unidentified hotspots, Mach/0.5 represents a substantial improvement in filesystem parallelism over Mach/0.2. We have already benefited from our incremental approach to parallelization by quickly bringing up a working system and then concentrating on parallelizing the worst bottlenecks first.

3.3.4 Future Work

Future filesystem parallelization enhancements will be guided chiefly by analysis of lock contention statistics to detect bottlenecks. Undoubtedly some of this work will focus on reducing *bfreelist_lock* contention as well as on improved inode and buffer locking. Selective use of inode read locks could dramatically increase parallelism on commonly-used files and directories

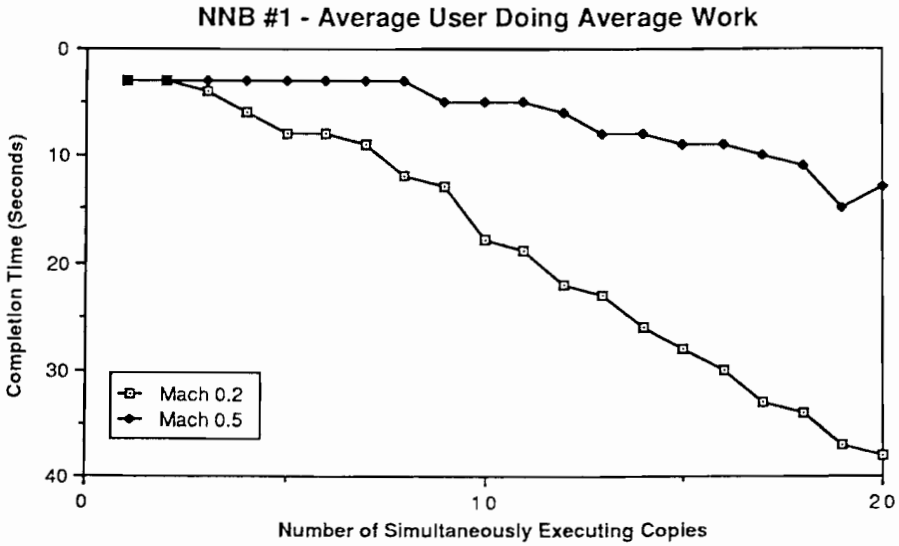

Figure 4: The Average User Working under Mach/0.2 and Mach/0.5

and could be achieved with small modifications to *namei*, *iget*, and *rwip*. An additional interface to the buffer cache could be provided for the case where a buffer is going to be read but not written. (*bread* must assume that the buffer will be modified by the caller.) In this case, the buffer cache could read-lock the buffer, allowing it to be shared by other readers.

More aggressive optimizations are conceivable. For example, inode locking as a means of preventing simultaneous overlapping modifications of file data largely could be eliminated. Buffer locking can synchronize modifications to the same block of file data. Inode locking could be restricted to the cases where the file's size would change or the I/O would span multiple file blocks. An optimization of this nature might have a beneficial effect on database operations against large, random-access files.

Finally, the direction of our work will change somewhat as we incorporate the latest CMU release of Mach, which contains a vnode layer and client and server NFS. This work is already well under way and has had a major impact on filesystem locking strategies.

4. Network Parallelization

Parallelization of the network subsystem was accomplished by dividing the network code into the same layers as defined by the ISO/OSI 7-layer model. Each layer, Link (device driver), Network (IP, ARP), and Transport/Session (TCP, UDP) was examined and parallelized separately. By so doing, we realized two benefits. First, multiple developers could work on separate sections of code with only minimal interference. Second, lock contention and overall performance could be examined and effort applied to only those algorithms or data structures revealed to be bottlenecks.

4.1 General Lock Policy

The network code presented a fundamental problem for parallelization: not only could data transfer be initiated by the local user but also asynchronously from the network. In other words, the user may send packets to the network interface whenever he wishes and (from the standpoint of the kernel) the network interface may send packets whenever it wishes. This behavior is different than that of the filesystem where interrupts do not generally represent unsolicited I/O operations but the completion of a user-initiated event.

Rather than poll the network interface for new packets, the 4.3BSD code, triggered by a network interrupt, pushes the packet across multiple protocol layers all the way up to the socket queue. In a kernel using locks to serialize simultaneous transactions, care must be taken to prevent the obvious deadlocks that can result from threads simultaneously traversing these layers in opposite directions.

To prevent deadlocks, permit multiprocessor execution, and encourage a speedy initial implementation, we decided upon a straightforward locking policy: each protocol would have a single, global lock guarding its data. A protocol's lock would be taken when using any associated protocol code and released when the protocol invoked a lower or higher layer. A thread that could not immediately acquire one of these locks would be put to sleep and awoken when the lock became available. This scheme was

sufficient for protocols such as ARP which have little traffic, but not acceptable for IP, TCP and UDP where there is significantly more traffic. For these "high-use" protocols, we ultimately developed finer-grained locking schemes on a per-connection basis.

The protocols we parallelized included TCP, UDP, ICMP, ARP and IP. We did not have the time or the need to parallelize other protocols present in the 4.3BSD distribution, such as Xerox NS or VMTP from Stanford.

A number of asynchronous kernel threads were created to handle timer based events for the various protocols. Under 4.3BSD all timer based operations, such as connection time-out, keep-alive transmission, and packet retransmission are performed at interrupt-level from the callout queue. As these actions may need to take locks, all such operations were moved into separate kernel threads.

4.2 Link Layer

The link layer primarily consists of device drivers. The Multimax uses intelligent controllers for all I/O operations, including Ethernet. Refer to Section 2.3 for the details of interaction with the Ethernet device driver.

4.3 Network layer

The network layer consists of the IP, ARP and ICMP protocols.

4.3.1 ARP

ARP packets are handled by two kernel threads with a single global lock around all ARP data structures. One of these threads processes incoming ARP packets; the second thread is used to time out old entries in the ARP table. While finer-grained locking has been considered, analysis of lock statistics shows that there is little lock contention in this area and we have concentrated our efforts elsewhere.

4.3.2 IP

The IP code is almost completely free of locks. Most packets pass through the IP layer without ever taking a lock. The major exception is packet fragmentation and reassembly, which is controlled by a single lock. On networks where there is a great deal of IP fragmentation, this single lock may be a bottleneck; however, with a single exception, on most local area networks there is no IP fragmentation. Even our Internet connection receives only an occasional IP fragment.

The addition of Network File System (NFS) functionality will create a greater need for IP fragmentation of UDP packets. Currently, Mach does not support NFS but when NFS support becomes available we will revisit the issue of IP fragmentation.

A separate kernel thread was created to handle IP timeouts. The only use of these timeouts is to remove old fragments from the queue. A thread was required as the IP lock needs to be held during this operation.

One interesting problem existed with incoming source routes. These are IP options to be used in replies to the incoming message. The original 4.3BSD implementation used a static structure to contain this information. As IP is a state-less protocol, there is no "connection" information maintained. A classic uniprocessor assumption was made that no other thread could change the data before the reply was sent.

With no per-connection structure to store this information, a place needed to be found to store the information. The solution used was to save the information in Mach's equivalent to the 4.3BSD *u-area*.

4.3.3 ICMP

The ICMP code is similar to IP in that few locks are required. In fact, the only lock is in the case of REDIRECT requests, i.e., changes to the route table. Management of the route table is described below.

4.3.4 Route Table

Routing information may be used by any network layer protocol. It is currently used by both IP and ICMP. Our analysis has shown that the routing data structures, while frequently used, did not warrant fine-grained locks. The reason for this is that the time spent within the routing code is relatively short. To provide for increased parallelism, the routing structures are protected by a read/write lock rather than a mutual exclusion lock.

The existing 4.3BSD code already had a reference count on the route table entries. This reference count is protected under lock and assures us that routing entries will not be unexpectedly deleted.

4.4 Transport/Session layer

The TCP and UDP protocols were parallelized in almost identical ways. For both of these protocols a linked list of all connections is maintained. In the Mach/0.5 implementation described in this paper, a mutual exclusion lock protects all operations to this list, including lookups. A new version of the kernel which uses read/write locks has already been implemented to allow simultaneous lookups.

To find the correct connection the global lock is taken prior to calling *in_pcblookup*(). Once the connection is found, a reference count in the per-connection *inpcb* structure is incremented (preventing the deallocation of the structure), the global lock is released and the *inpcb* lock acquired, thereby guarding the connection against simultaneous access. This lock is held during all packet processing. This lock also implicitly protects the *tcpcb* or *udpcb* structure pointed to by the *inpcb*, as appropriate. While it may be possible to release the lock, or to use a read/write lock, current statistics do not suggest that such a change is warranted.

In addition to the reference count added to the *inpcb*, another flag was added for protocols such as TCP to indicate that the connection is being closed. This field was necessary to prevent race conditions, for example, further transmission attempts while closing the connection.

The single major difference between TCP and UDP is that TCP provides reliable data transfer. This implies the need for retransmission, maintaining connections, etc. Much of this activity is driven from two timers; "fast" (200ms) and "slow" (500ms). As the TCP connection chain must be traversed during these timeouts and locks taken, separate kernel threads were created to handle each of these timeouts.

The 4.3BSD code uses the *callout* queue to implement timeouts. Having the entry in the callout queue awaken the timeout threads would have worked, however, it would also require that timeout routines be rewritten as threads. To work around this limitation, two additional threads were created, *pffast_thread* and *pfslow_thread*, which call the per-protocol timeout functions. Thus, an implementation could either single stream timeout functions, or wake additional threads for increased parallelism. In our current implementation, all of our timeout functions are implemented using separate threads, providing greater parallelism.

4.5 Miscellaneous

The user layer and protocol layer are quite separate in the 4.3BSD model. The user layer interacts through system calls such as *read*(2), *write*(2), *send*(2), and *recv*(2). Each of these calls ultimately uses a *socket* structure, each of which now has its own lock. All operations on the socket are protected by this lock. When the user sends data, the data is chained to the socket while the socket lock is held. Receive operations dequeue data from the socket, also under lock. Lower level protocols that work with sockets, such as TCP and UDP, must not only take the relevant *inpcb* lock but any appropriate socket locks as well.

The network memory pool is almost exclusively made up of *mbufs*, which come from two pools, the mbuf list and the cluster list. *mbufs* may be allocated or deallocated in both interrupt and thread context, so each list has its own simple lock. Although *mbufs* are used widely in the 4.3BSD code, the implementation simply required adding locking calls to a few macros and subroutines. One significant change was creating threads to allocate

additional memory when needed. These threads permit blocking during mbuf and cluster memory allocation.

Under 4.3BSD UNIX pipes use sockets for I/O. Connecting two sockets together required a significant amount of work to avoid deadlock when attempting to take the two socket locks. A solution similar to the *dup2* problem was used here – socket pairs were always locked by taking the lock of the lowest addressed socket first. With only this exception, the remainder of the network parallelization allowed pipes to operate in parallel as well.

4.6 Parallelized Network Calls

The network parallelization effort allowed a large number of 4.3BSD calls to execute in parallel and permitted outgoing and incoming packets to be handled on any processor. As with the filesystem code, a few calls were heavily used and the remainder were parallelized because they shared data structures with the performance-sensitive routines.

4.7 Network Performance Analysis

There are many components within the network subsystem that affect performance. While we would have liked to measure the performance of individual pieces of the network code, for our purposes here we present an analysis based on total TCP throughput. Unfortunately, there are no standard network performance tests similar to the disk I/O tests performed by the Neal Nelson Benchmarks. Therefore, we constructed our own network performance tests.

The fundamental test we developed creates a TCP connection to a remote system and repeatedly sends data using the *write*(2) system call. The recipient simply reads and discards the data. The size of the write requests was varied using values of 1, 2, 10, 64, 100, 512, 1000, 2000, and 16K bytes. During the development of these tests we experimented with other values but did not find that they yielded much additional information. The total amount of data sent was controlled so that the length of the test was at least five seconds and ran no more than ten minutes. These times were chosen to provide steady-state performance without forcing

the benchmarking process to become needlessly lengthy. Only time to transfer the data was counted; time to establish and close the connection was not included. For each request size the experiment was repeated three times and the average of the three runs was used in the accompanying graphs.

The test just described uses only a single TCP connection. We created another test using multiple copies of the single-stream test. Data was also collected while running 2, 3, 5 and 10 simultaneous copies. As before, the multiple connection experiments were run three times and the average of the three runs was used.

The systems used to run these tests were two Multimax-320 systems, each configured as follows:

- 4 APC-01 CPU boards, 2 two-MIPS NS32332 CPUs per card, total 16 MIPS
- 5 SMC-16 memory cards, at 16 megabytes of memory, total 80 megabytes
- 1 EMC-I, with one Ethernet interface and one masstore interface
- 1 CDC Sabre disk drive
- Private Ethernet connection between these two machines

Baseline measurements were taken using the Mach/0.2 "serial" kernel (see Figure 5). For each request size from one through 512 bytes there was almost no increase in aggregate throughput when the number of connections was increased. Aggregate throughput only increased with additional connections when the request size exceeded 1000 bytes, and then by only 17% (1000 byte requests) to 42.5% (16K byte requests). As expected, the *master* CPU, forced to process all interrupts and incoming packets, as well as TCP, IP, and ARP requests was limited in the amount of network traffic it could handle. The performance improvement observed with larger packets resulted from the amortization of the (fixed-size) TCP/IP packet overhead across a larger quantity of data. Analysis of the Mach/0.5 aggregate throughput (see Figure 6) shows that increasing the number of connections increases the aggregate throughput. For example, when making 1000 byte requests (typical for FTP) two simultaneous connections had 83% additional throughput over a single stream; obviously, the theoretical

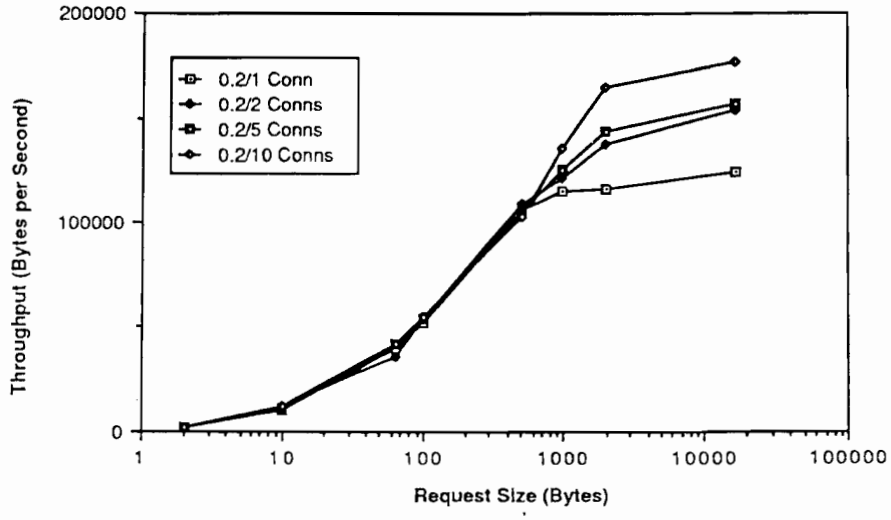

Figure 5: Mach/0.2 Network Performance

maximum would be 100%. Ten simultaneous connections had 517% additional throughput. Many multi-processor benchmarks attempt to attain linear speedup as the number of simultaneous tasks increase. While this goal also applies to benchmarks of network performance on a multi-processor, additional constraints prevent the network subsystem from achieving linear speedup. The speed of the transmission line represents an absolute maximum on network throughput regardless of the number of

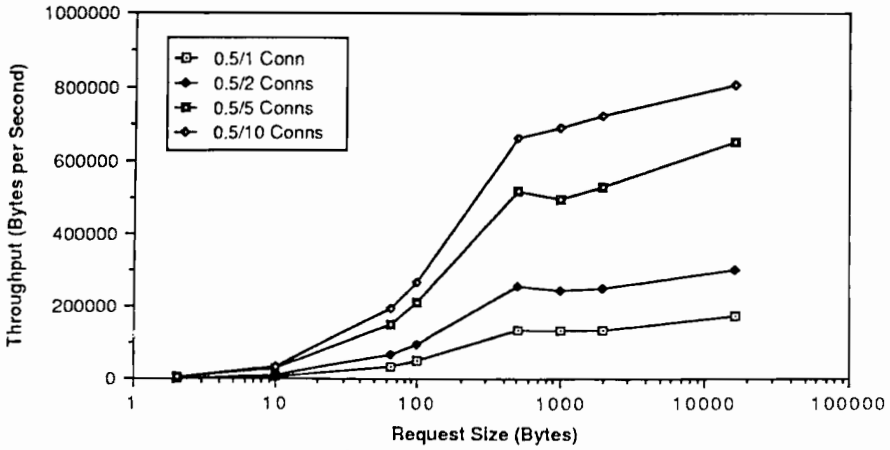

Figure 6: Mach/0.5 Network Performance

processors used. Unbounded linear speedup, in this case, is not possible. Our tests were run using standard 10M bit/second Ethernet. The maximum theoretical data throughput of 1.25M bytes/second does not take into account TCP header, IP header, source and destination address, CRC bytes, preamble, and collisions. In addition, the TCP protocol also requires acknowledgments from the receiver, each of these requiring a 64 byte packet. Given all of this, the effective maximum transfer rate is much closer to 1 Million bytes per second. The tests described in this paper show a maximum throughput of approximately 803,000 bytes per second, with every sign that additional connections could be supported, further increasing throughput.

As we have mentioned, the design of the network parallelization was done under a framework where separate functional areas of the network, such as IP, ARP, TCP and UDP were all parallelized separately. For the most part, changes in one area were not dependent upon another. We analyzed performance and lock contention in these separate areas and optimized only those areas which would yield the greatest payoff. An example of this occurred between version Mach/0.4 and Mach/0.5. Figures 7 and 8 show performance results for the serial and two parallel versions of Mach. Mach/0.4 contained a global lock around the TCP subsystem and another around the IP subsystem. Mach/0.5 removed

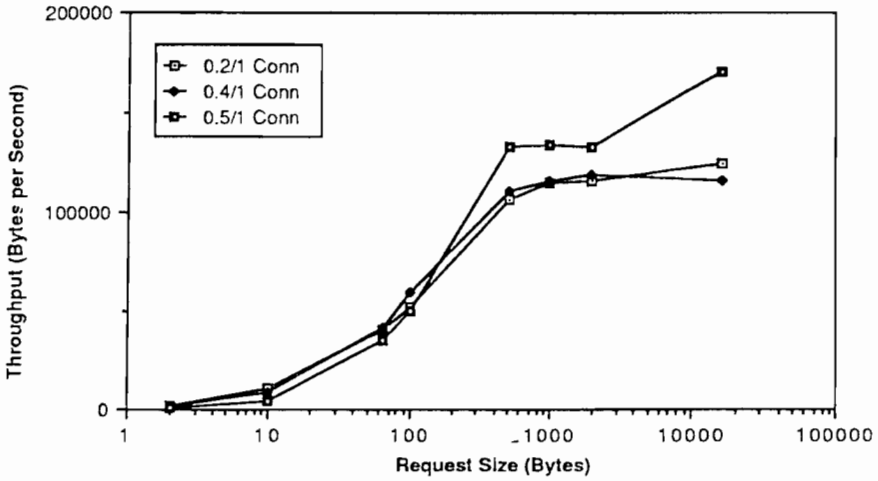

Figure 7: Single-Stream Performance, Mach/0.2 vs. Mach/0.5

the IP lock completely; the only locking done within the IP layer is around the fragmentation/reassembly queues. In addition, the global lock around TCP was removed in favor of a per-connection lock. Analysis, design and implementation of these changes were accomplished over a two-month time span. The increased performance, especially with multiple connections, is obvious from the graphs. Modern computer systems require ever increasing performance from their networking facilities. Network subsystem performance is crucial on the Encore Multimax, which depends on an Ethernet interface for all user terminal traffic. Parallelization of the network code has significantly enhanced multi-stream TCP performance.

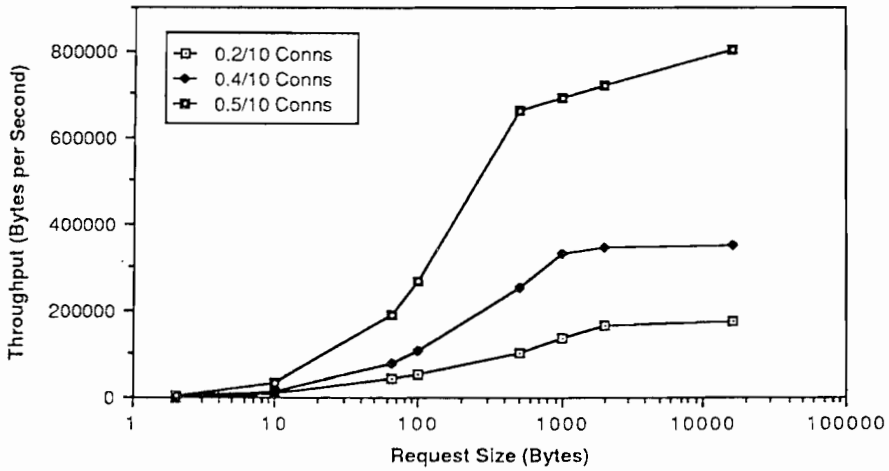

Figure 8: Aggregate Performance Gained by Incremental Parallelization

5. *Debugging*

Encore has created a number of tools to assist in the debugging of multiprocessor kernels. First, our standard user-level, high-level language debugger has been modified slightly to understand remote kernel debugging. All Encore operating system kernels include a very low-level, nearly stand-alone debugging module that understands how to observe and control the execution of the larger kernel. This debugging module communicates over a serial line with a production machine running our high-level debugger.

The module permits single-stepping, tracing and observation of the activities of any processor on the machine being debugged. The high-level debugger allows the user to control the target kernel at the level of C statements or assembly-language instructions. In fact, the very same debugging module and high-level debugger are used to debug our low-level firmware and diagnostic code. Needless to say, these tools are invaluable.

For our project, we also developed a standard approach to coding locks. All locks are coded as macros, so the developer may modify a single definition to include extra debugging code or even, on occasion, to change the type of lock being used. A single, compile-time option indicates whether extra lock debugging code is to be included in the kernel image. Another compile-time option causes the locking routines to record statistics about lock contention rates.

When compiled for lock debugging, the lock routines themselves record the program counter where the lock was locked and unlocked but only for mutual exclusion locks, which is why many of our locks start out as mutual exclusion locks and are changed to read/write locks after being debugged. The lock routines also record lock ownership and check whether locks are being re-taken by the same owner or being released without having first been acquired (two common errors). Note that the locking routines will always record lock ownership, regardless of compile-time options. Lock ownership is a valuable clue when analyzing crash dumps.

Frequently, a function will include at its beginning debugging assertions about the state of various relevant locks. Especially important are assertions about locks that are expected to have already been taken by another routine. Such assertions prevent the vexing problem of unruly threads clobbering unlocked data. If any of these assertions fail, the kernel panics.

The blocking lock routines optionally track interesting lock statistics, including number of attempts, misses, forced reschedules, minimum and maximum wait times, and total time threads spent waiting. Similar statistics have recently been added to *simplelock*s.

These statistics can be retrieved and displayed at any time with a simple user-level utility, allowing us to dynamically monitor a running system to detect locks with high contention rates

under varying workloads. This tool has been quite useful in guiding our parallelization efforts.

6. Summary

The data demonstrate that Mach/0.5 is significantly more parallel than Mach/0.2 in terms of filesystem and network performance. We have a framework in place for incrementally increasing the parallelism of the operating system.

We have reason to believe that current Mach/0.5 performance is competitive with commercial operating systems for tightly-coupled parallel architectures. A benchmark developed and run at CMU compared the performance of Mach/0.5, running on a Multimax-320 using 2-MIPS NS32332 processors, to that of another vendor's commercial operating system running on 4-MIPS Intel 386 processors [Rashid 1989]. Single-stream, the benchmark completed half as quickly on the Multimax. By ten streams, however, the Multimax completed the benchmark more quickly than the system built on faster processors.

Our efforts to minimize source code modifications and to always `#ifdef` the modifications we made are paying off today as we merge our filesystem and network changes with CMU's latest enhancements, including new networking features and a vnode layer for the filesystem.

Future work will focus on further improving the parallelization of Mach/0.5's 4.3BSD compatibility code. In particular, remaining frequently used or long-running system calls will be targeted for parallelization. Signal-related system calls are now at the top of our list. There are a number of other calls that only require *unix_master* because they depend on updating one or two 4.3BSD data structures (e.g., the *proc* table) that are maintained chiefly for the benefit of user-level utilities that read kernel memory. In particular, *fork*(2) and *exit*(2) fall into this category.

Mach/0.5 was released in August, 1989 to the twenty-five Encore customers already running an earlier version of the parallelized filesystem and network code. The current release, Mach/0.5.3, includes enhancements such as TCP and UDP read/write locks described within this paper.

References

M. Bach and S. Buroff, Multiprocessor UNIX Operating Systems, *AT&T Bell Laboratories Technical Journal* 63, pages 1733-1749, October 1984.

J. Barton and J. Wagner, Beyond Threads: Resource Sharing in UNIX, In *Winter 1988 USENIX Conference Proceedings*.

J. Boykin and A Langerman, The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis, In *Workshop Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 105-126, 1989.

G. Hamilton and D. Code, An Experimental Symmetric Multiprocessor Ultrix Kernel, In *Conference Proceedings, 1988 Winter USENIX Technical Conference*, 1988.

A. Langerman, J. Boykin, S. LoVerso, and S. Mangalat, A Highly-Parallelized Mach-based Vnode Filesystem, In *Conference Proceedings, 1990 Winter USENIX Technical Conference*, pages 297-312.

[NNB] Neal Nelson and Associates, Neal Nelson Benchmark Report, 1986. Benchmark results reprinted by permission.

R. Rashid, Threads of a New System, *UNIX Review*, August 1986.

R. F. Rashid, A Proposal to UNIX International to Integrate Mach Technology into UNIX System V, May 1989. Submission to UNIX International Multiprocessor Working Group.

U. Sinkewicz, A Strategy for SMP ULTRIX, In *Conference Proceedings, 1988 Summer USENIX Technical Conference*, pages 203-212.