Performance Analysis of 64-Bit Capriccio

Shaun Verch

May 20, 2012

1 Introduction

Capriccio[1] is a lightweight userspace threading library written to run on 32-bit Linux. The authors cited performance improvements over pthreads, and attempted to make Cappriccio a drop in replacement for pthreads. The original Capriccio implementation for the most part matched the pthreads API, but there were a few barriers to adoption.

1. It was only available as a static library, which meant that the application using Capriccio had to be specially compiled to use it
2. It required special compiler modifications to work around the limitations of a 32-bit address space. These compiler modifications allowed dynamic stack allocation, which added additional complexity and overhead.

To work around these issues, I converted Capriccio to a dynamically linked library and ported it to 64-bit. This paper is an analysis of the performance of 64-Bit Capriccio compared to the performance of 64-Bit pthreads.

2 Architecture Overview

2.1 Pthreads

Pthreads has one kernel thread per user thread. This effectively means that the kernel controls everything about thread management.

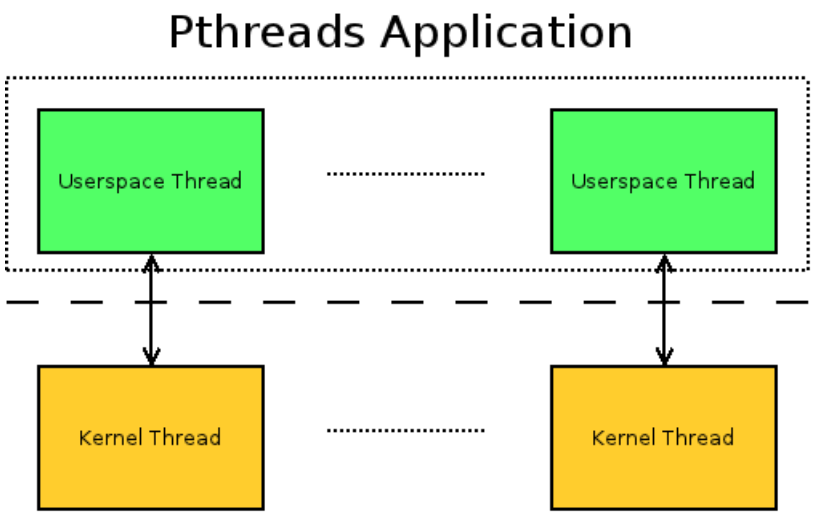

Figure 2.1.1: Pthreads Architecture

[1] http://capriccio.cs.berkeley.edu/

2.2 Capriccio

Capriccio currently has only one kernel thread and one user thread for all the thread management. All Capriccio threads are run in the context of this thread.

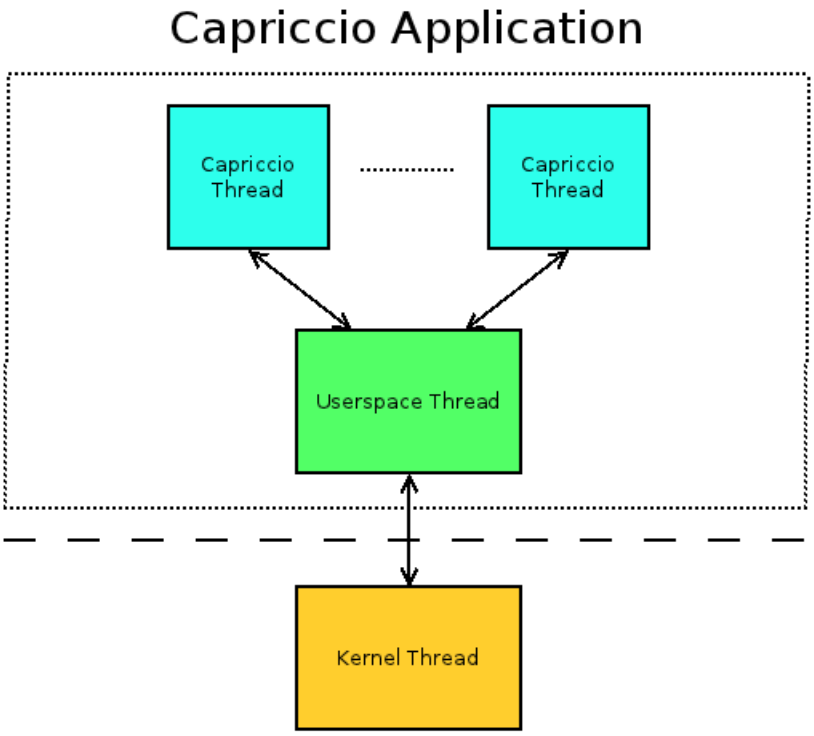

Figure 2.2.1: Capriccio Architecture

Because there is only one kernel thread in Capriccio, it can never be allowed to block. This means that I/O must be handled in a special way to prevent this from happening. Capriccio manages this by intercepting system calls that would block in the kernel, putting the currently running thread to sleep, and dispatching the I/O itself using some other mechanism that will not block the main thread. These mechanisms all have some performance penalties compared to blocking I/O, which means that pthreads may have better performance on I/O heavy loads. This is described in more detail below.

3 Scheduling

3.1 Pthreads

Since there is a one to one mapping between user threads and kernel threads in pthreads, the scheduling is entirely done by the kernel. This means that threads can be preempted by the kernel at any time without any of this information reaching user space.

3.2 Capriccio

In Capriccio, all the scheduling decisions are made in user space. A small coroutine library is used to manage thread context switching. Scheduling in Capriccio is cooperative and can be done either by having a scheduler thread or by having the yielding thread call the reschedule function directly. Capriccio instruments a number of function calls, and inserts yield points there. Currently there does not exist any mechanism for preempting threads.

A user space scheduler such as Capriccio has a few major advantages.

3.2.1 Context Switching

For pthreads, every context switch requires a trap to the kernel. For Capriccio, a context switch is as simple as a return from a function call, and is in fact implemented this way. This means that Capriccio has much less overhead than pthreads for switching between threads.

3.2.2 Scheduler Sandboxing

When using a user space scheduler is that it is much easier to tune the scheduling algorithm to match the needs of the specific application without unintentionally affecting performance of other things on the system. Capriccio supports a number of different scheduling algorithms which can be set by simply setting an enviroment variable before running the application.

3.2.3 System Stability

Since the kernel is not involved in scheduling decisions, the kernel scheduler does not get overloaded when Capriccio is being used. Since Capriccio does not require a large number of kernel threads, the system overall remains responsive and stable even when Capriccio is being stress tested. This is not the case for pthreads.

3.2.4 Mutual Exclusion

Mutual exclusion can be very expensive when kernel involvement is required. The NTPL implementation of pthreads on Linux attempts to work around this

by using futexes, which operate mostly in user space and only cause a trap to the kernel when a thread needs to sleep because a lock is already taken. Since Capriccio is single threaded and uses cooperative scheduling, mutual exclusion is trivial. Even if support were added for preemption and multiple processors, mutual exclusion would still never require a trap to the kernel even when a thread must block. This has strictly less overhead than any mutual exclusion strategy that the NTPL implementation of pthreads could use.

4 Stack Management

4.1 Pthreads

Since every pthread has a kernel thread associated with it, each pthread effectively must have two stacks. In current verisions of Linux, each kernel stack is 4K[2]. This is only one additional page of overhead as far as memory usage, but it also requires additional bookkeeping. Pthreads also seems to have some performance issues because of its stack allocation, but this is discussed later.

4.2 Capriccio

The only stack needed for each Capriccio thread is a user space stack. The stacks can be allocated in slabs or individually using mmap or malloc. Different considerations had to be made when dealing with a 32 bit system versus a 64 bit system.

4.2.1 32 Bits

Because the virtual address space on a 32 bit machine is only 2^{32} bytes, it can only support up to 512 threads with 8MB stacks, even assuming the address space is entirely filled with only thread stacks. Even if the stack size is reduced to 1MB, the number of threads that can be created is still only 4096. To work around this, the authors of Capriccio developed a dynamic stack allocation strategy. This involved creating a callgraph, annotating each node of the callgraph with the stack usage of the associated function, and inserting checkpoints in such a way that a checkpoint would be reached before the stack overflowed on any path. This algorithm has a few major disadvantages.

1. It requires special compiler modifications. Gcc shows some support for dumping a callgraph, but it does not appear to be intended for this purpose. In addition, if the checkpoints are written in C code, then the compiler must effectively be run twice. Once to link everything and generate the callgraph, and once to compile with the checkpoints inserted.

2. The program must be recompiled to use Capriccio. The original version of Capriccio was a static library presumably for this reason.

[2] http://kerneltrap.org/Linux/Defaulting_To_4K_Stacks

3. It does not handle stack usage in precompiled libraries without special annotation or conservative defaults.

4. It does not address variable length arrays and uses of alloca().

5. It introduces additional overhead.

4.2.2 64 Bits

Stack management is greatly simplified on a 64 bit machine. Because the address space on a 64 bit machine is 2^{32} times larger than the address space on a 32 bit machine, it can support as many thread stacks as needed without any special modifications. Because allocating a region of virtual memory does not immediately allocate the memory itself, we effectively get on demand stack allocation for free from the VM subsystem.
There could be wasted memory if we have long running threads that sometimes require a large stack. However, this can be solved by either periodically recycling the threads, or by recycling the stack. This is actually possible because the stack pointer, along with all the other context information, is available to Capriccio.

5 I/O management

The I/O management for Capriccio revolves around not blocking the currently running thread. To this end, any calls that would block on I/O are intercepted by Capriccio and the request is dispatched to whatever I/O management scheme is currently registered.

5.1 Registration

Capriccio uses the following interface to register new socket io schemes, where *IO* is replaced by the specific io scheme using a C macro. A similar method is used for disk io registration.

int sockio_*IO*_is_available: This is checked during io registration to make sure the system actually supports the io scheme we are trying to use.

int sockio_*IO*_add_request(iorequest_t *req): This is used to register any request for io on a descriptor. For sockets, calls to accept also go through this interface.

void sockio_*IO*_init(): Do any initialization that may be required for this io scheme.

void sockio_*IO*_poll(long long usec): Poll the underlying I/O scheme to see if any descriptors have activity. For any successfully completed I/O requests, wake up the threads that were waiting by putting them on the run queue.

5.2 Poll

When poll is used for the underlying I/O management and a request is registered, Capriccio first attempts the syscall directly after setting all the I/O to be non blocking. If this attempt fails with the error code of EWOULDBLOCK, the request is added to a list of pending requests. Then, on each call to the I/O poll routine, the normal poll syscall is used to check for activity on any of the registered file descriptors. The main disadvantage of poll is that it requires copying all the file descriptors that are being monitored for activity to the kernel on each call, which may be a bottleneck if a large number of connections and files are open. However, in some cases this is not an issue. For example, when a large number of threads call accept on a single socket, only that one file descriptor has to be polled for activitiy.

5.3 Worker threads

This is a scheme for managing disk I/O. Capriccio creates a pool of real threads to handle disk operations. These threads can then perform blocking I/O on behalf of the main thread, which allows the main thread to continue running. This is an example of one situation where Capriccio may use additional kernel threads.

5.4 Epoll

In order to avoid copying a large array of file descriptors on each poll call, the 32 bit version of Capriccio originally used epoll as an underlying I/O scheme. Epoll was designed specifically to avoid the unnecessary copying of poll.

5.4.1 Epoll Functionality

The kqueue api has three function calls associated with it:

epoll_create: The epoll_create system call creates a new epoll structure in the kernel and returns a file descriptor that can be used to interact with it.

epoll_ctl: The epoll_ctl system call is used to control the epoll structure in the kernel. It is used to register file descriptors that are to be monitored.

epoll_wait: The epoll_wait system call waits for activity on any of the registered descriptors, and returns event structs for descriptors that have had activity.

This scales better than poll because it only requires one copy of the file descriptor to request notification when activity occurs, and one copy each time there is activity on the file descriptor. For a poll driven interface, this is the minimum amount of information that must be exchanged between the kernel and user space.

5.5 Kqueue

Before porting Capriccio to 64 bit, I ported the 32 bit version to FreeBSD. FreeBSD does not support epoll, but does have kqueue[3], which also attempts to solve the performance issues associated with poll.

5.5.1 Kqueue Functionality

The kqueue api has two function calls associated with it:

kqueue: The kqueue system call creates a new kernel event queue and returns a descriptor that can be used in subsequent calls to kevent.

kevent: The kevent call provides the bulk of the functionality of the kqueue interface. It is used to both register events and report activity on the registered events back to the user. Arrays of kevent structs are used to register or return events. There are various fields in the kevent struct that specify what kind of event to watch for, and what action to take.

Kqueue has the same amount of copying as epoll, so it effectively solves the same problem. However, there is one additional optimization that can be done. Since the same syscall is used for registration and notification, an effectively free poll call can be done each time new I/O is registered.

5.6 Performance Penalties

All of these strategies require some form of active polling on the part of Capriccio. This means that in addition to the actual I/O syscall, there is the added overhead of all the polling that must be done to whichever I/O scheme is chosen. Since I/O heavy loads require traps to the kernel anyway, pthreads may have better performance in this case because of the extra layer of indirection required by Capriccio.

6 Testing Strategy

6.1 One thread per connection

Capriccio is designed to fit the "one thread per connection" model. This is a useful programming model for writing a server, because the code that actually deals with clients can be written as if it is only dealing with one client and thus be greatly simplified. The code that actually creates the threads and accepts connections can be the same for every server using this model. Thus, writing a server just amounts to writing code describing what should be done for each client.

[3] people.freebsd.org/ jlemon/papers/kqueue.pdf

6.2 Setup

My goal in testing was not to test computationally intensive server work. Instead I focused specifically on how the number of connections affected performance to see if the one thread per connection model is viable. To this end, I simulate "load" on the server side by introducing a delay before the server responds. This is implemented as a sleep, so does not take processor cycles, but it does keep the connection open longer. To simulate a large number of clients making requests to the server, I created a small number of multithreaded clients with each thread making requests. For each test run, I did the following.

1. Start the server running on either Capriccio or pthreads, passing in as parameters the number of threads to create and the amount of time to sleep during each connection.

2. Start enough clients to have one client thread per server thread. To reduce the number of client instances that have to be run and analyzed, the clients are themselves multithreaded. The clients were able to handle at least 2000 threads each, and were not taken past this to avoid having the clients be the bottleneck

3. Collect vmstat data on the server side to determine how well the server is handling the load

4. Collect connection statistics on the clients to determine the responsiveness of the server

6.3 Computationally Intensive Loads

Computationally intensive loads are not tested here. Because Capriccio only has a single kernel thread, we expect its performance to be far below that of pthreads for this type of load on multiprocessor systems. Support can be added for multiple processors, but this will require additional overhead due to synchronization.

7 Results

7.1 Requests Per Second

Capriccio and pthreads both seem to handle requests in bursts. This section does not give much insight into the relative performance, but it does demonstrate differences in stability and consistency. Pthreads in general shows less predictable behavior. In addition, the bursts in pthreads have a much higher amplitude, both in the lows and the highs.

7.1.1 Capriccio

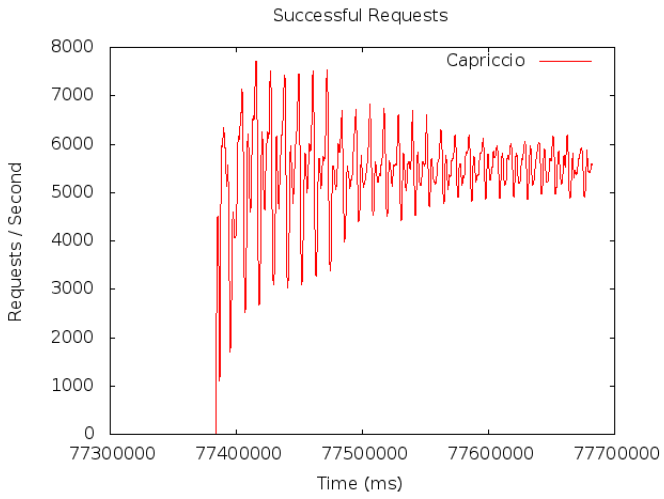

Figure 7.1.1: Capriccio running 15000 threads

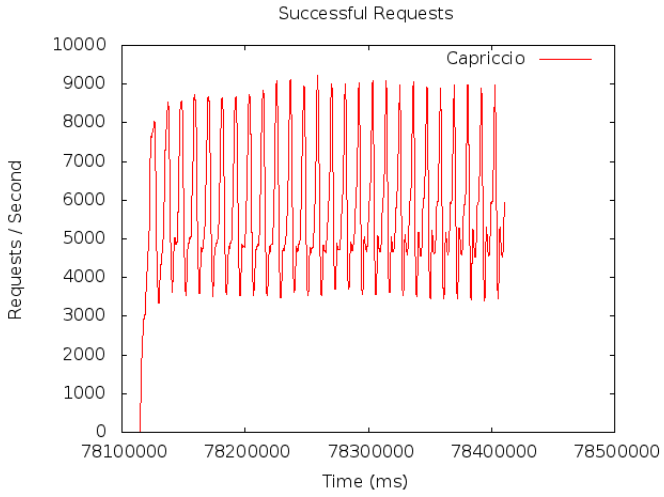

Figure 7.1.2: Capriccio running 25000 threads

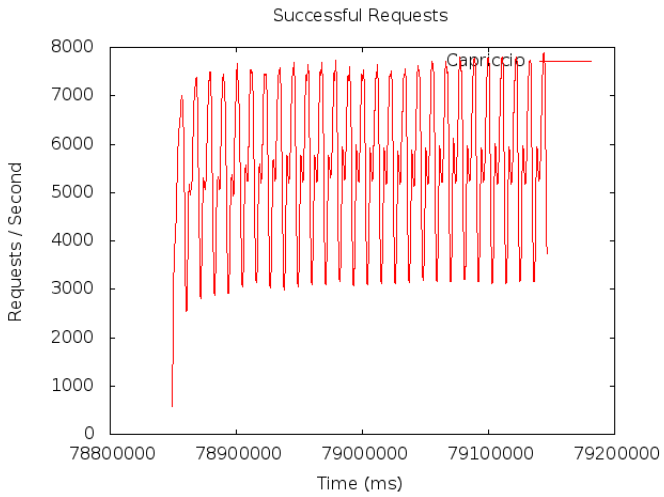

Figure 7.1.3: Capriccio running 30000 threads

7.1.2 Pthreads

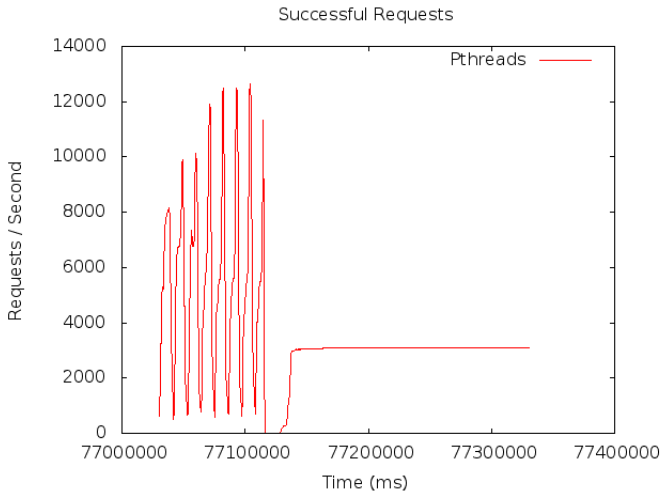

Figure 7.1.4: Pthreads running 15000 threads

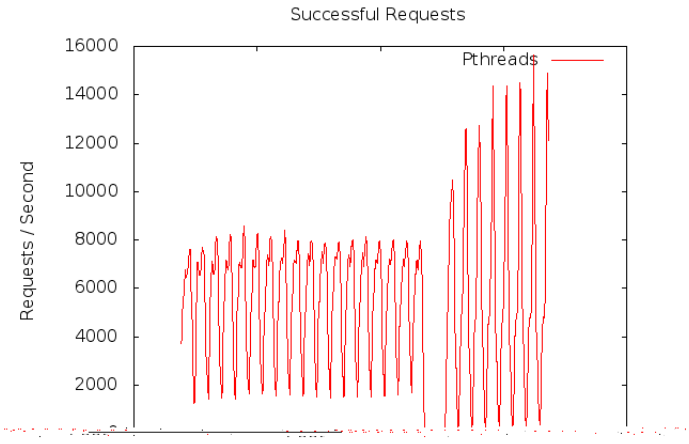

Figure 7.1.5: Pthreads running 25000 threads

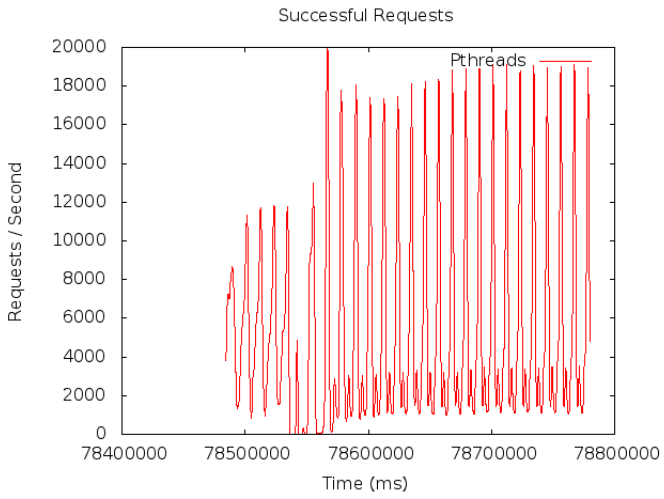

Figure 7.1.6: Pthreads running 30000 threads

7.2 Open Connections

The number of open connections also seems to follow the same burst pattern. This seems to indicate that the burst pattern shown in the previous section may be caused by something network related. For example, the connect timeout may be a factor here, because the server may just discard some client connections while it is too busy, and the clients may take a while to notice. The fact that the threads are in lock step as far as opening and closing connections are concerned is still not completely explained by this. Note that the server timeout is only one second, which does not match the period of this burst pattern.

7.2.1 Capriccio

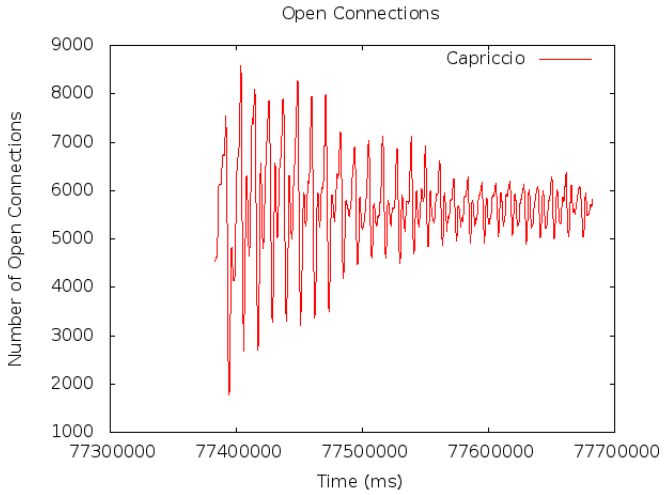

Figure 7.2.1: Capriccio running 15000 threads

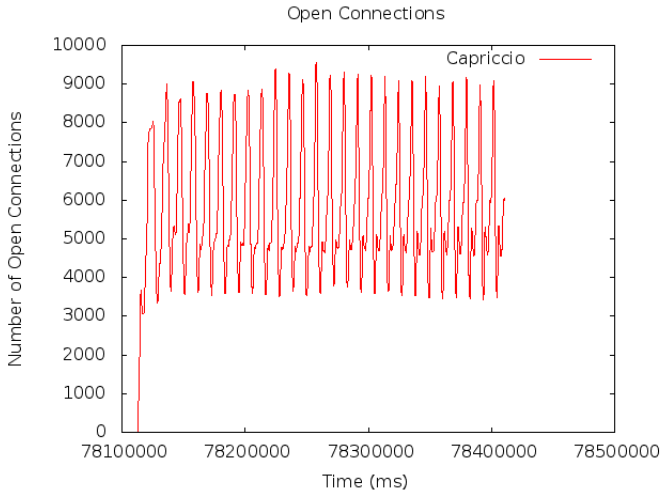

Figure 7.2.2: Capriccio running 25000 threads

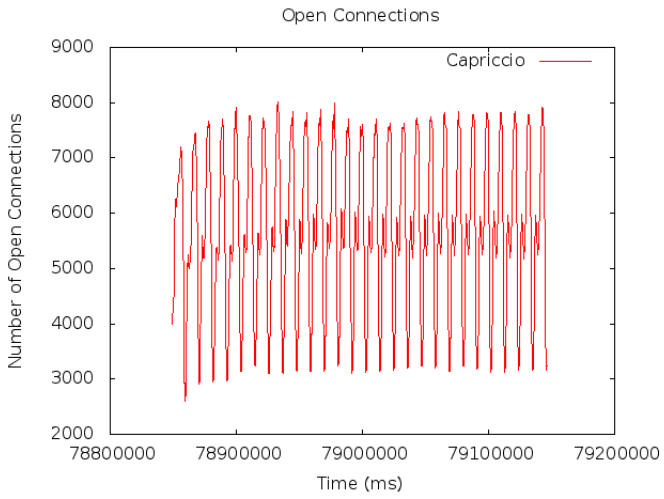

Figure 7.2.3: Capriccio running 30000 threads

7.2.2 Pthreads

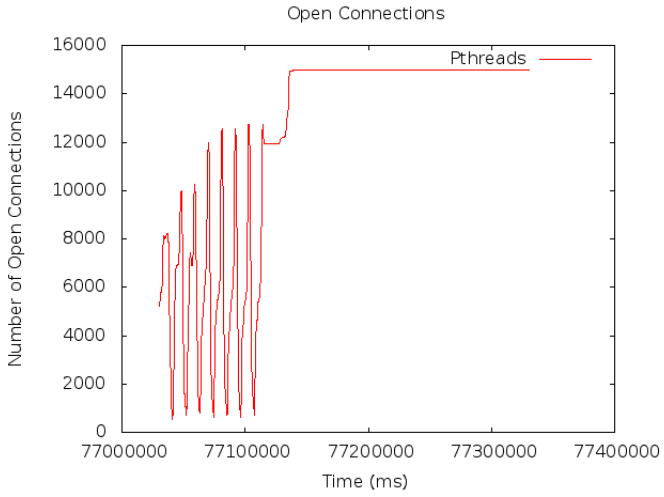

Figure 7.2.4: Pthreads running 15000 threads

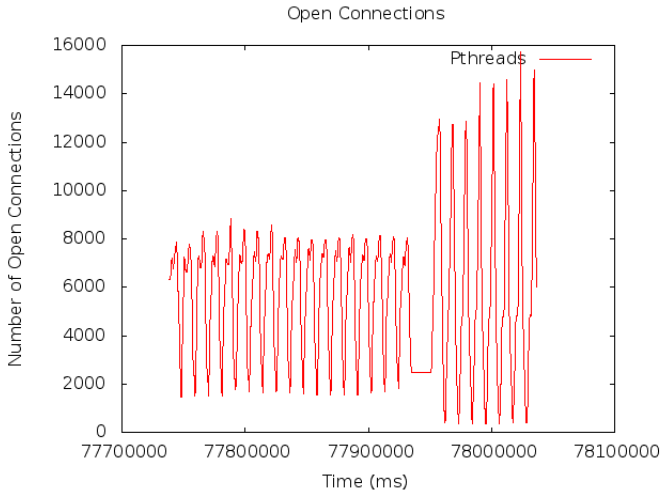

Figure 7.2.5: Pthreads running 25000 threads

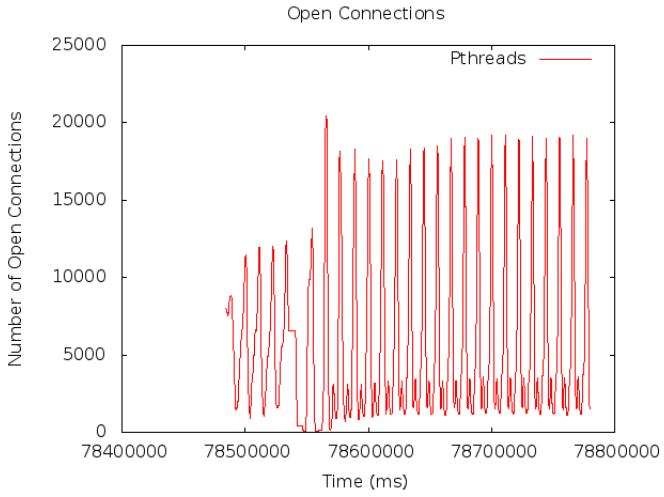

Figure 7.2.6: Pthreads running 30000 threads

7.3 Total Throughput

Despite the differences in consistency and the burst behavior, Capriccio and Pthreads demonstrate similar throughput on thread counts below 30000. As mentioned previously, the tests favor Capriccio due to the fact that the load is not CPU intensive and Capriccio is only running on a single thread. However, this does indicate that if Capriccio were modified to run on multiple processors, it could keep up with pthreads at least for thread counts below 30000.

7.3.1 Capriccio

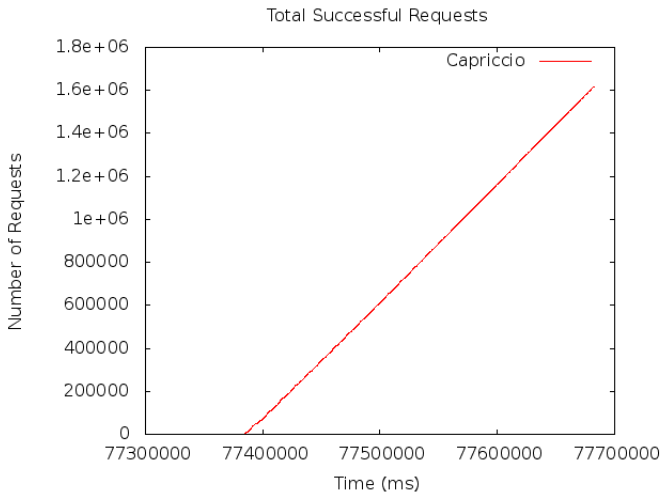

Figure 7.3.1: Capriccio running 15000 threads

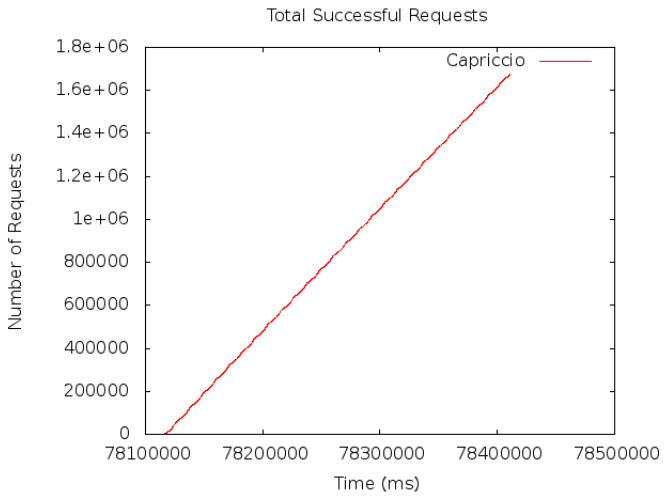

Figure 7.3.2: Capriccio running 25000 threads

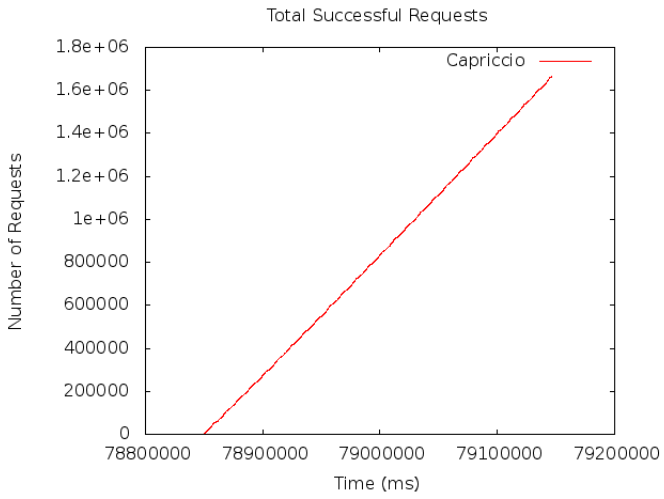

Figure 7.3.3: Capriccio running 30000 threads

7.3.2 Pthreads

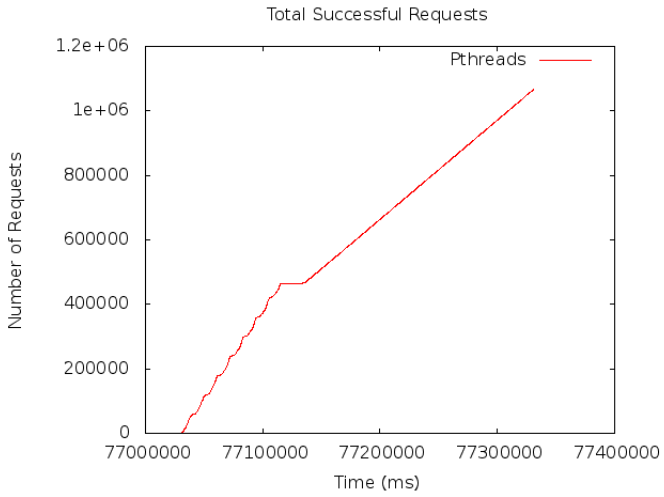

Figure 7.3.4: Pthreads running 15000 threads

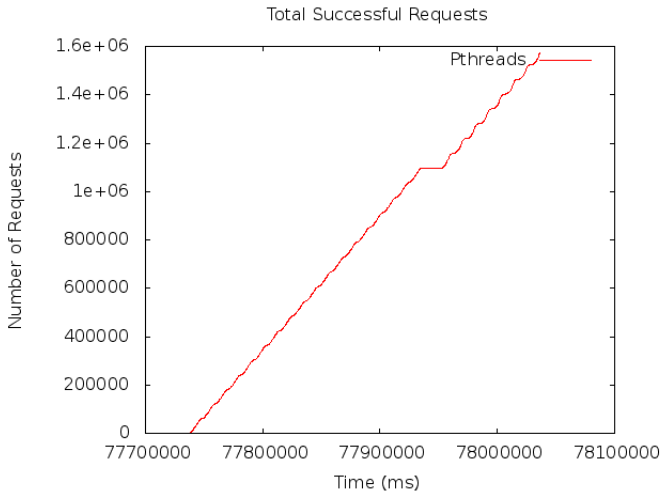

Figure 7.3.5: Pthreads running 25000 threads

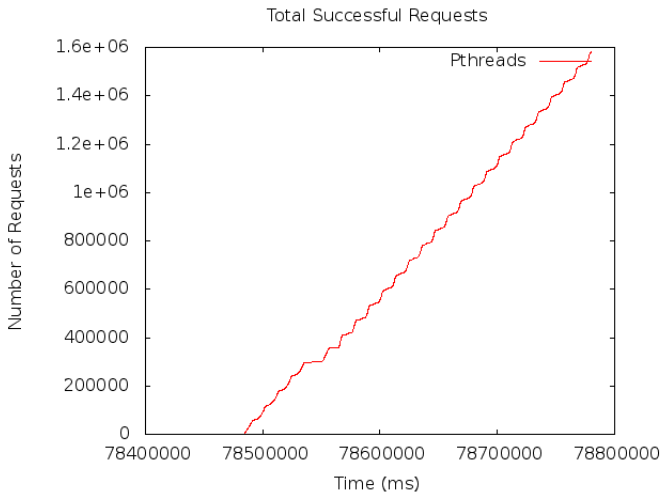

Figure 7.3.6: Pthreads running 30000 threads

7.4 Pthreads Thread Creation

When creating a large number of threads (over 78900), pthreads had severe performance issues. This came up before any connections were even opened (but did effect performance here as well). From the vmstat logs, when creating over about 78900 threads with 8MB stacks, we see a lot of pages being swapped in and out. The number of threads that can be created before the swapping starts are affected by the stack size. This seems to indicate that pthreads is not only allocating the stack for each thread, but is also touching all the pages somehow. It is possible that pthread_create zeroes out each new thread's stack, or that it touches pages on each new thread's stack to avoid overhead from allocation while the thread is running. Capriccio does not touch the new stacks, which means that the stacks are allocated on demand by the VM subsystem. The fact that memory is only allocated on Capriccio when it is needed is why Capriccio can still run with over 100,000 threads while pthreads cannot (and in fact causes instability in the host system).

7.5 Capriccio Limits

7.5.1 Number of threads

Capriccio could create up to around 130000 threads with 1GB stacks before malloc failed. With 1MB stacks Capriccio could create around 300000 threads, but at that point it started swapping out memory, possibly because just the part of the stack being used became a limiting factor.

The highest thread count tests for Capriccio were with 108000 threads, because this is already past what pthreads can do and is on the order of magnitude of the testing resources available (with 54 lab machines and a maximum of 2000 threads each, that allows for 108000 client threads).

Note that the throughput does indeed suffer with this number of threads. This seems to indicate that some part of Capriccio or the system as the whole is not scalable.

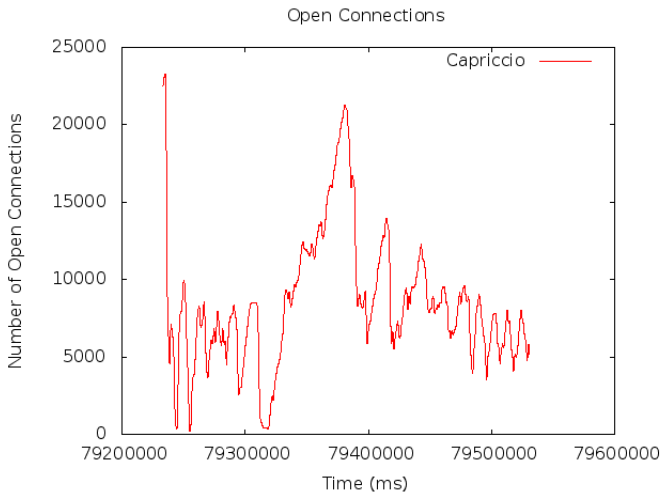

Figure 7.5.1: 108000 Threads

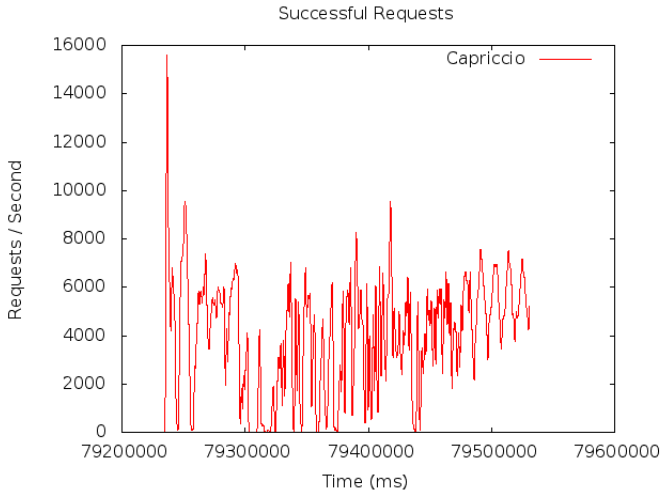

Figure 7.5.2: 108000 Threads

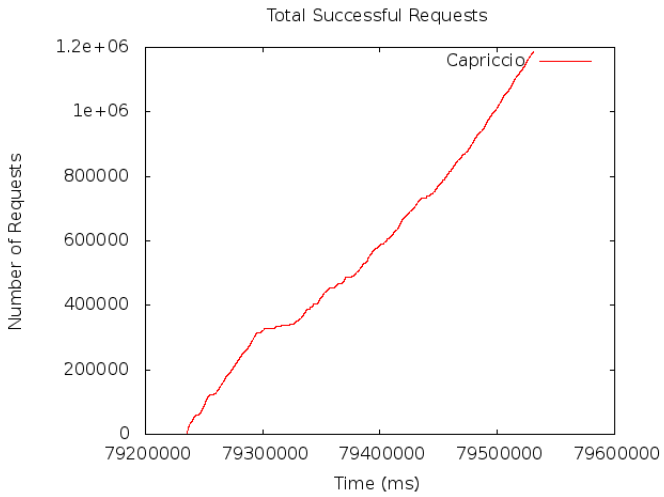

Figure 7.5.3: 108000 Threads

7.5.2 Number of open connections

On tests with 108000 threads, the number of open connections did not match the number of threads. It seemed that increasing the delay on the server increased the number of open connections, but that had a limit. The rate at which new connections are made seems to slow down over time.

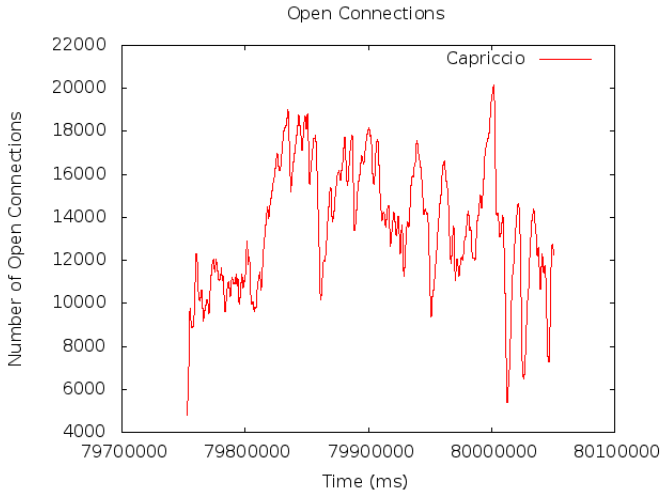

Figure 7.5.4: 2 sec delay

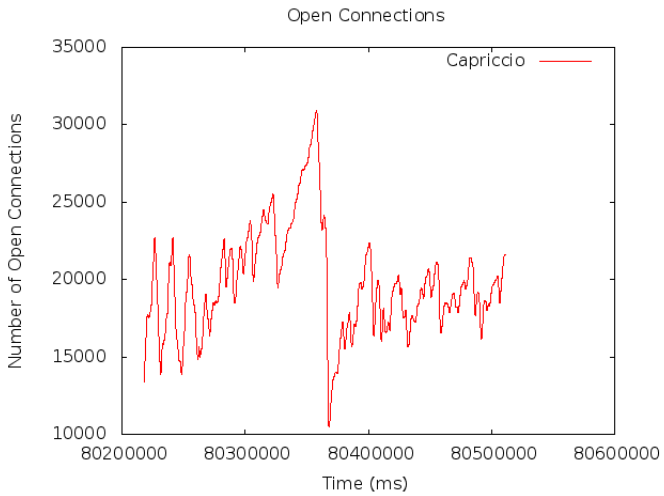

Figure 7.5.5: 4 sec delay

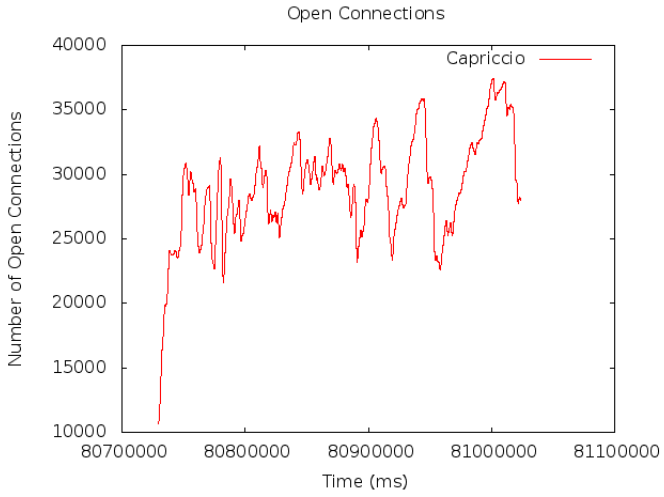

Figure 7.5.6: 8 sec delay

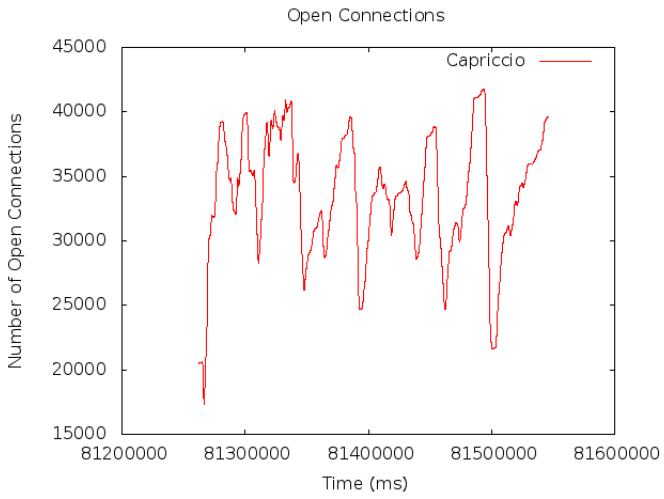

Figure 7.5.7: 16 sec delay

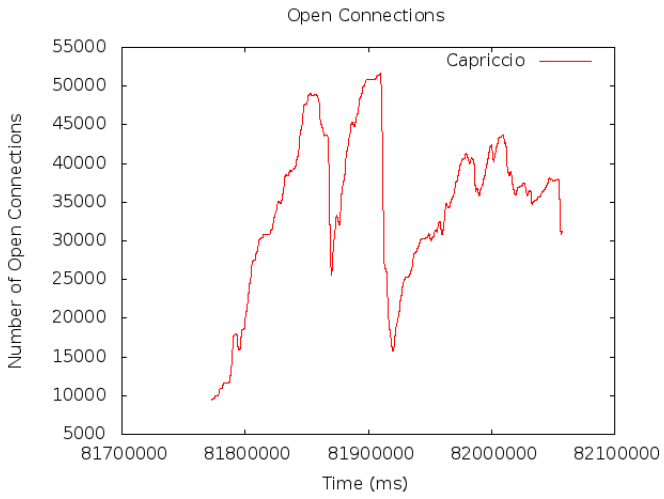

Figure 7.5.8: 32 sec delay

8 Future Work

8.1 Scheduler Activations

One major issue with the current implementation of Capriccio is that it only uses a single kernel visible thread. This means that it cannot execute on multiple processors. One possible way to address this would be to use Scheduler Activations[4]. This is a mechanism to allow the kernel to give some information about its scheduling decisions to the user space scheduler. The user space scheduler can then be aware of how many running kernel threads it has available to it, and act accordingly.

8.2 Scheduler Algorithms

As mentioned previously, the fact that the scheduling is done entirely in user space means that different scheduling algorithms could be used without affecting the rest of the system. This would allow for schedulers that are tuned to match the needs of specific applications. For example, a scheduler could be made that prioritizes threads making write calls on a socket as opposed to threads making read calls, increasing responsiveness for connections that are already open.

[4]http://people.freebsd.org/ deischen/docs/Scheduler.pdf

8.3 Preemption

Because Capriccio has no mechanism to preempt threads, a user thread can monopolize all the processor time that Capriccio has by sitting in a busy loop and making no calls that Capriccio can intercept and force the thread to yield. To prevent this, preemption must be added. However, it should only be used on misbehaving threads that have not yielded in a while if it requires any kind of kernel intervention such as a timer interrupt because preempting in this manner too frequently will introduce additional overhead.

8.4 I/O Schemes

Since different I/O schemes can be easily swapped out, there is room for experimentation. The problem with the current I/O methods employed by Capriccio is that they do not get an instantaneous response. They all require something resembling a poll on active file descriptors, with the exception of the worker thread scheme. One solution may be to have a dedicated thread just for dealing with I/O rather than polling whenever Capriccio happens to be in control. An interrupt driven scheme may be an option, but it may incur additional overhead.

9 Conclusion

The results shown here seem to indicate that user space threading is a viable option, especially on 64 bit systems where stack management is much simpler. These results came from Capriccio using poll as the underlying I/O scheduling mechanism and only one thread. Better results may be obtained by improving the I/O management mechanism, the scheduling algorithm, and the multiprocessor support of Capriccio.
It would be worth looking into why pthreads is touching so many pages on thread creation to see if this can be safely avoided. If it can be, it may be a way to make pthreads in its current form scale better to large numbers of threads.

A Capriccio 64 bit port

The main issues that came up in porting Capriccio to 64 bit were the parts of the code written in x86 assembly. This included the time stamp counter and the coroutine library.

A.1 Time Stamp Counter

The time stamp counter loads its value into two special registers which had to be dealt with differently on a 64 bit machine. Despite the fact that the registers on a 64 bit machine can hold the whole time stamp value, the x86 rdtsc instruction still splits it into two different registers.

A.2 Coroutine Library

The coroutine library was the biggest change in the port to 64 bit. Because it had to manipulate the stack in an architecture dependent way on the first call to a coroutine, the co_call function was written entirely in assembly. Rewriting the function entirely in 64 bit assebly caused gcc to fail with relocation errors. Therefore, it is currently running with a mixture of C code and assembly in the co_call routine.

B Syscall wrapping

Certain syscalls must be intercepted when using Capriccio because of the fact that the main thread cannot block on I/O. The syscall wrapping originally used some aliasing that only worked for the static version of the library. After converting the library to a dynamically linked library, I intercepted syscalls by linking capriccio before libc and using dlsym to call the original syscalls when needed. Later I started using the gcc linker's "-wrap" option. This causes Any undefined reference to *symbol* to be resolved to __wrap_*symbol* and any undefined reference to __real_*symbol* to be resolved to *symbol*.

C System Configuration

To run a large number of threads with pthreads, I had to change a lot of system parameters, including the max thread limit and the number of allowed virtual memory mappings. Pthreads also puts a strain on kernel memory because of all the kernel stacks. Since all of Capriccio's threads are handled internally by Capriccio, these system wide limits do not require modification.