The following paper was originally published in the
Proceedings of the USENIX MACH III Symposium
Santa Fe, New Mexico, April 1993.

Using Continuations to Build a User-Level Threads Library

Randall Dean

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Using Continuations to Build a User-Level Threads Library

Randall W. Dean

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

(412) 268-7654

Internet: rwd@cs.cmu.edu
Fax: (412) 681-5739

Abstract

We have designed and built a user-level threads library that uses *continuations* for transfers of control. The use of *continuations* reduces the amount of state that needs to be saved and restored at context switch time thereby reducing the instruction count in the critical sections. Our multiprocessor contention benchmarks indicate that this reduction and the use of *Busy Spinning*, *Busy Waiting* and *Spin Polling* increases throughput by as much as 75% on a multiprocessor. In addition, flattening the locking hierarchy reduces context switch latency by 5% to 49% on both uniprocessors and multiprocessors. This paper describes the library's design and compares its overall performance characteristics to the existing implementation.

1. Introduction

Mach 3.0 [8] has been available since 1990 for anonymous FTP as a platform for building operating systems and embedded applications. To support this development environment, a library that implements multiple threads of control and mutual exclusion is provided. In Mach 2.5 [1] C-Threads [6] is the library that supplies these needs. The Mach 2.5 implementation maps user threads one-to-one with Mach kernel threads resulting in extensive usage of kernel data structures and expensive kernel context switches. The old Mach 3.0 implementation solves the one-to-one mapping problem by multiplexing user threads onto equal or fewer kernel threads. The old implementation uses a complex locking hierarchy suitable for multiprocessors that requires a small state machine inside the context switch critical section. Combined with the need to acquire multiple locks and save full user register state, this approach results in a relatively high latency for user context switching.

The need to save full register state at context switch time also affects throughput on multiprocessors. Regardless of how finely grained the locking hierarchy is, it is necessary to hold a single central lock during a context switch to protect the blocking computation from being resumed by another thread. The time needed to save and restore the user registers becomes the lower bound on critical section size.

We have applied *continuations* to C-Threads to address the effects on latency and throughput caused by long critical sections. Continuations have existed in the programming language community as an abstraction for describing a future computation for many years [14]. They have also been used in the Mach 3.0 kernel [7] as a basis for control transfers between execution

contexts. Their use in describing control transfers in the kernel reduced context switching times and kernel stack utilization.

In C-Threads, continuations have not only directly reduced critical section size, but have enabled further optimizations. The direct effect of applying continuations is a reduction of the number of words saved and restored at context switch time. By building a continuation when a potentially blocking C-Thread call is made, the thread state needed at context switch time can be described with only 2 words. The building of the user continuation can be done upon entering the C-Threads library so no lock needs to be held. This reduction in critical section size makes it possible to flatten the locking hierarchy which further reduces the size of the critical section. The net result is a reduction in latency of 5% to 49% on our uniprocessor context switch benchmark. These changes along with *Busy Waiting*, *Busy Spinning*, and *Spin Polling* increase throughput by as much as 75% as measured by our multiprocessor contention benchmark.

Considerable research has been done in the area of user-level thread packages. Of particular relevance to the work described in this paper is the work on scheduler activations [2] [3] and adaptive spinning strategies [12]. We believe there would be synergistic benefits from the integration of these efforts.

Section 2 gives an overview of the C-Threads library interface and describes the relevant internal interfaces and implementations. Section 3 describes how continuations were implemented and used with the new version of C-Threads. Section 4 describes the optimizations used in implementing the new version. Section 5 compares latency and throughput in the old and new versions of C-Threads and shows how certain optimizations effected throughput.

2. C-Thread Internals

C-Threads [6] was originally designed as a threading library that transparently used Mach features to supply parallelism. Implementations included a version that used a Mach task for each C-Thread, a version that used a single thread in a single task to supply coroutines, and a version that used one Mach task with a Mach thread per C-Thread. The old Mach 3.0 version of the library multiplexes C-Threads onto equal or fewer kernel threads because of concerns about kernel data structure use such as kernel stacks and kernel context switching times.

The remainder of this section outlines the C-Thread external interface. It also describes enough of the old Mach 3.0 implementation's internal interfaces and structure to provide a basis for understanding the changes made in the new version.

2.1. Interface

A detailed description of the C-Threads interface can be found in the CMU technical report by Cooper and Draves [6]. This section briefly reviews the interface described in that document. Where additions to the interface were made, motivation and a more detailed description of the calls is given.

2.1.1. Threads

C-Threads are created and managed with the following calls:

- **cthread_t** *cthread_fork***(cthread_fn_t func, void *argument)** creates a C-Thread that executes function **func** with argument **arg**.
- **void** *cthread_exit***(void *status)** exits the current C-Thread returning **status**.
- **void *** *cthread_join***(cthread_t thread)** waits for C-Thread **thread** to terminate and returns the exit **status** of that thread.

2.1.2. Mutexes

C-Threads provides a mutual exclusion primitive called mutex that guarantees at most one holder. Mutexes are manipulated by the following calls:

- *mutex_lock***(mutex_t m)** Acquires the mutex **m**.
- *mutex_unlock***(mutex_t m)** releases the mutex **m**.

2.1.3. Condition Variables

Condition variables allow one thread to wait for another thread to signal that event or change in state occurred. The following calls handle condition variables:

- **void** *condition_wait***(condition_t c, mutex_t m)** waits for condition **c** releasing mutex **m** while waiting. When the condition is signaled, the mutex is reacquired before *condition_wait*() returns.
- *condition_signal***(condition_t c)** signals condition **c** waking up at least one waiting thread.

2.1.4. Kernel Threads

The old C-Thread library multiplexes multiple C-Threads onto equal or fewer kernel threads. The best performance is achieved when pre-emptive kernel context switching is minimized without losing parallelism. Two new interface calls were added to allow the application designer to control the number of kernel threads used:

- **void** *cthread_set_kernel_limit***(int limit)** sets the maximum number of kernel threads that can be created. A value of zero indicates that there is no limit.
- **int** *cthread_kernel_limit***(void)** returns the current maximum number of kernel threads.

2.1.5. Thread Wiring

The C-Threads library includes a mechanism to allow a user to disable C-Threads multiplexing on a per C-Thread basis:

- **void** *cthread_wire***(void)** binds a C-Thread to its underlying kernel thread.
- **void** *cthread_unwire***(void)** removes the binding making it possible for multiplexing to occur.

Once a thread is *wired* it becomes possible for the user to make Mach thread calls such as *thread_priority*(), *thread_assign*(), or *thread_info*() on the kernel thread and have the calls only affect the associated C-Thread. Programs can use this facility to run specialized threads at elevated priority, for example, the BSD 4.3 Single Server [8] uses this for threads that interact with kernel devices.

2.2. Old Implementations

The old Mach 3.0 implementation draws from the Mach 2.5 coroutine and threads versions. The coroutine version multiplexes multiple C-Threads onto a single Mach kernel thread. This requires machine dependent code for user context switching, per-object queues, and a global queue of runnable threads called the *run_queue*. With only one kernel thread there can be no contention for the internal data structures so no locking is required in the coroutine version. The threads version binds C-Threads to kernel threads and has as many kernel threads as there are C-Threads. Per-object queues must now be locked because of pre-emptive scheduling, but no machine dependent context switching code or global *run_queue* is needed since this is all handled by the kernel.

From the Mach 2.5 implementations was taken all of the queues, all of the locking, the multiple kernel threads, and the machine dependent context switching code. With these two sources also came an internal programming layer called *cprocs*. *Cprocs* provides an implementation independent virtual processor layer that is used to implement the higher level C-Threads primitives. The differences between the various C-Threads implementations are hidden within this layer. The remainder of this section will describe interfaces and implementation details that reside below the *cproc* layer.

2.2.1. Locking

Locking internal C-Threads data structures cannot be handled with mutexes since they are implemented by the library. Instead, C-Threads does its locking with *spin_lock*s. *Spin_lock*s are implemented using the hardware's native test-and-set, compare-and-swap, or exchange-memory instruction. On uniprocessors that do not supply such an instruction, a kernel-supported software restartable-atomic-sequence is used [4]. In order to minimize bus contention when locking a *spin_lock*, we implement them as a test followed by the atomic operation. This keeps most of the spinning in the cache.

For each mutex and condition variable there is a queue for C-Threads that are blocked waiting on that object. There is also a global *run_queue* on which a C-Thread is enqueued when it becomes runnable. Each of these queues is protected by a different *spin_lock*. The *spin_lock* guarding the *run_queue* is called the *run_lock*.

2.2.2. Blocking and Resuming

The routine invoked when a C-Thread blocks is called *cproc_block*(). This routine does one of two things depending on whether or not the blocking thread is *wired*. If the thread is wired, it blocks by calling *mach_msg*(), waiting for a message to wake it up indicating that it can run again. If the thread is not *wired*, it checks the *run_queue* for another thread to run. If there is a C-Thread present, it contexts switches to that thread.

If a thread is not present, then the underlying kernel thread must disassociate itself from the blocking C-Thread and become idle. Since the stack that the kernel thread uses in user space is associated with the C-Thread and not the kernel thread, the first thing necessary is to acquire a stack to use. It is not possible to use the blocking C-Thread's stack, since the blocking thread could become runnable and start running on another kernel thread. A pool of idle C-Threads with small stacks is kept for this purpose. A C-Thread is pulled from this pool or created if necessary. The blocking C-Thread then context switches to this idle C-Thread invoking the routine

cproc_waiting(). *Cproc_waiting*() simply sits in a loop calling *mach_msg*() waiting for a message to wake it up. When it receives a message, it acquires the *run_lock* and checks for C-Thread. If a C-Thread is found, its execution will be resumed. If not, the idle thread loops back to *mach_msg*().

When an operation such as *condition_signal*() or *mutex_unlock*() makes a C-Thread runnable again, *cproc_ready*() is invoked. This routine inserts the newly runnable thread on the *run_queue* and wakes up any idle kernel threads.

2.2.3. The State Machine

A small state machine is used to keep track of each C-Thread. A thread can be either RUNNING, BLOCKED, or SWITCHING. The first two states have the obvious meaning and are and easy to deal with in *cproc_ready*() and *cproc_block*(). The third state, SWITCHING, occurs when a thread is in the process of blocking and has released the lock on the queue of the object for which it is waiting.

2.3. Evaluation of Old Implementation

While the old Mach 3.0 implementation is a great improvement over the Mach 2.5 implementations by enabling user level context switching, we felt that further enhancements could be made. The old implementation is very aggressive about blocking. Spinning before blocking has been shown to be superior to just spinning or just blocking [12]. Mach 3.0 has better scheduling primitives for synchronization then *mach_msg*() [5]. Finally, the old Mach 3.0 code is complex and hard to maintain. We created the new implementation to address our concerns with the shortcomings of the old implementation.

3. Continuations

A continuation is an object that fully describes a future computation. A continuation can be built, invoked, and passed around as an argument like any other object. The Mach 3.0 kernel has two types of continuations. The first type of continuation describes the user's state and is built when the thread crosses the protection boundary from user space to kernel mode. The second type describes the state that has accumulated since entering the kernel and is built when the thread blocks inside the kernel.

As with the kernel implementation, there are two distinct continuations in the new version of the C-Threads library. A continuation that describes the user's computation at the time a C-Thread routine is called is built upon entrance to the library. We call this the *User Continuation*. The other continuation present in this implementation is used internally to describe the computation that should occur when a C-Thread is resumed after blocking. We call this the *Internal Continuation*.

We have two different continuations because, like the kernel, the computations they describe are so different. The *User Continuation* describes the user state that has accumulated up to the point of a C-Threads operation being invoked. This is potentially large and difficult to examine, because it resides on the user's stack. Furthermore, the user's state is machine dependent. We build the *User Continuation* to encapsulate this machine dependent state as a machine independent object. The *Internal Continuation* is a small, easily examined, easily manipulated, machine independent object consisting of a function pointer and an argument.

3.1. Building and Invoking Continuations

The kernel has a distinct protection boundary that makes it easy to build and invoke user continuations when the boundary is crossed. To make it possible for C-Threads to easily build user continuations, we supply a wrapper to potentially blocking C-Thread calls that acts much like the user-to-kernel protection boundary crossover. Building the *User Continuation* is done without holding any locks. The continuation is built by saving all the callee saved user registers on the current stack and saving the resulting stack pointer. Upon exit from the C-Threads library, this continuation is invoked. This entails setting the stack pointer to that saved at continuation creation, restoring the user's registers, and returning to the user routine which originally called C-Threads.

With the user's state encapsulated in the *User Continuation*, all that is needed to describe the state of the library is a single function pointer and argument. The *Internal Continuation*'s two words are saved inside the *User Continuation* in space reserved for this purpose. When this thread is resumed and its continuation is invoked, the function is called with the specified argument. The thread's stack pointer is derived from the address of the previously saved *User Continuation*.

3.2. The *filter* Mechanism

To centralize the manipulation of continuations, a single object called a *filter* is used. All manipulations of continuations within C-Threads are done through this object. A *filter* is a per-thread specified function that is invoked each time the manipulation of a continuation is necessary. A *filter* takes as a parameter the type of manipulation necessary. These are:

- FILTER_USER_BUILD: This is invoked by the wrapper of a potentially blocking C-Thread routine. It calls the real function, which is passed to it as an argument.
- FILTER_INTERNAL_BUILD: This is invoked when a C-Thread actually blocks. It is passed the function and argument that must be invoked when the C-Thread continues. It calls the FILTER_INTERNAL_INVOKE of the C-Thread that is being resumed.
- FILTER_INTERNAL_INVOKE: This is invoked by a blocking C-Thread from FILTER_INTERNAL_BUILD. It invokes the routine and argument saved in the *Internal Continuation*.
- FILTER_USER_INVOKE: This is invoked upon leaving C-Threads after entering with FILTER_USER_BUILD. It invokes the *User Continuation*.
- FILTER_PREPARE: This is invoked when C-Threads creates a new C-Thread. It is passed a routine and argument like FILTER_INTERNAL_BUILD that will be invoked when the created C-Thread begins running. This is essentially the same as FILTER_INTERNAL_BUILD without the invocation of FILTER_INTERNAL_INVOKE upon exit.

Filters can be defined outside of C-Threads. The *filter* described in the section, the *default_filter*, is the *filter* specified for a thread when it is created. The following two calls allow the *filter* for a C-Thread to be examined and changed:

- **cthread_fn_t** *cthread_filter***(cthread_t thread)** returns the current filter for C-Thread **thread**.

- **void** *cthread_set_filter***(cthread_t thread, cthread_fn_t func)** sets the current *filter* for C-Thread **thread** to **func**.

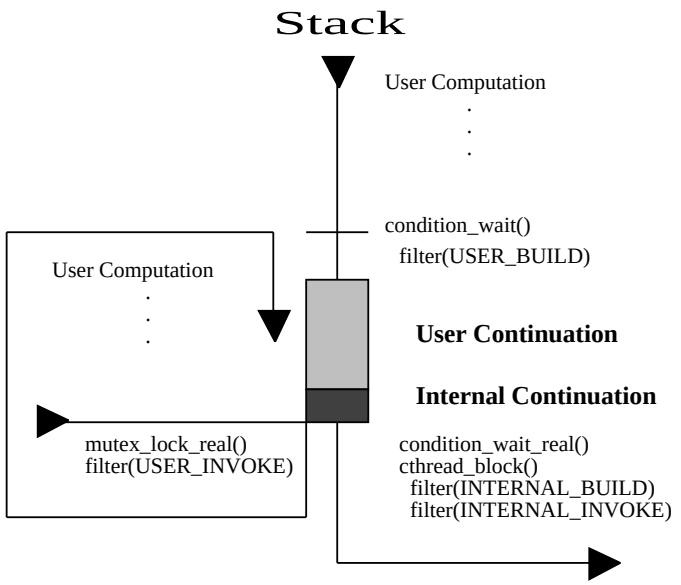

Figure 3-1: Example of user blocking with *condition_wait*()

3.3. Example of Continuation Use

Figure 3-1 shows the path taken as a user computation blocks with *condition_wait*(). The user calls *condition_wait*() which is a wrapper that invokes the *filter* to build the *User Continuation*. The *User Continuation* and space for the *Internal Continuation* appear next on the stack as built by the *filter*. The real *condition_wait*() is then called and the C-Thread blocks. When it blocks it invokes the *filter* again to build the *Internal Continuation*. This is saved in the space previously allocated for it when the *User Continuation* was built. The blocking thread then invokes the *Internal Continuation* of the C-Thread to which it is switching, removing the kernel thread from the blocking C-Thread and leaving it blocked.

When the blocked thread is resumed by another thread invoking the blocked thread's *Internal Continuation*, *mutex_lock*() is called to reacquire the mutex released when it blocked. Assuming this succeeds, the *User Continuation* is invoked and the user's computation continues with the stack back where it originally was when *condition_wait*() was first called.

4. Optimizations

Continuations enabled a number of changes to the new implementation of C-Threads. These include a flattening of the locking hierarchy and various sorts of spinning when the computation is no longer able to make progress. The optimizations that require spinning occur only on multiprocessors and assume either sequentially consistent [13] or processor consistent [9] memory. This section describes these changes and the motivations behind them.

4.1. Locking Techniques

The old implementation uses separate *spin_lock*s to protect the *run_queue* and the internal queue of each mutex and condition. The old implementation also requires a state machine to keep track of C-Thread state as threads are enqueued and dequeued and locks are acquired, released and reacquired. By using only a single *spin_lock* to protect all the queues and the internal state, we eliminate the need for the SWITCHING state of the state machine. Once the SWITCHING state is removed, there is no longer a reason to have the state machine at all.

While the use of only one *spin_lock* creates a single point of contention, it also makes it easier to add contention reducing optimizations. With this change and the use of continuations, the critical section of the context switch path actually becomes shorter than in the old implementation.

4.2. *Busy Spinning* and *Busy Waiting*

Busy Spinning is defined as spinning until some global state changes when there may be other work that the processor could accomplish. Spinning before blocking has been shown to be superior to only spinning or only blocking [12]. The new implementation spins for a constant number of iterations before taking action to yield the processor. The number of iterations to spin is currently a static, machine-dependent value determined at compile time. *Busy Spinning* currently occurs whenever a *spin_lock* must be taken and in *mutex_lock*() idle threads. We only *Busy Spin* on multiprocessors, since spinning would guarantee the obstruction of work on uniprocessors.

In *mutex_lock*() the thread spins attempting to acquire the mutex. If the mutex is acquired, the thread returns by invoking the user continuation. Otherwise, after spinning for the maximum number of iterations and not acquiring the mutex, the thread then attempts to acquire the *run_lock*, queue the mutex, and block.

An idle thread spins on the *run_queue* without holding the *run_lock* waiting for a C-Thread to appear that can be run. If this occurs, the idle thread attempts to acquire the *run_lock*, dequeue a C-Thread, and switch to it. After spinning for the maximum number of iterations and not having a C-Thread appear, the idle thread depresses its priority and begins again [5]. This priority depression yields the processor to any runnable kernel thread. When there are no runnable kernel threads except the depressed thread, the system automatically aborts the priority depression and resumes the idle thread. We call this depressed priority spinning or *Busy Waiting*. C-Threads that are *wired* also use *Busy Waiting*.

Unfortunately, *Busy Waiting* will not work perfectly on a machine with other tasks running if these tasks have compute bound threads. Once an idle thread depresses its priority, it would never run again if there were another thread on the system running. To address this, we use a timeout when we depress priority in the general *spin_lock* case so that starvation does not occur. For the idle thread we keep a queue of the kernel threads currently idle, and when we insert a thread on the *run_queue*, we call *thread_depress_abort*(). This aborts the priority depression and returns the thread to its original priority.

The old Mach 3.0 C-Threads implementation uses *mach_msg*() instead of *Busy Waiting* in the idle thread. When the *run_queue* is empty the idle thread waits to receive a message indicating

that a C-Thread has been inserted in the queue. This disadvantage of this approach is that it is not possible for the idle thread to act on the newly runnable C-Thread until it receives a message. The C-Thread languishes in the *run_queue* for the duration of a message send followed by a receive. It is also the case that sending a message and receiving a message are considerably costlier operations than depressing priority and aborting it.

4.3. *Spin Polling*

In both of the previously described specific cases of *Busy Spinning*, one of the termination conditions of spinning results in attempting to acquire the *run_lock*. Since the *run_lock* is the only *spin_lock* protecting state used internally by C-Threads, in a finely grained parallel application contention is likely. With *mutex_lock*() this occurs after a failure to acquire the mutex. With an idle thread, it occurs when a C-Thread appears in the *run_queue*. In both of these cases, instead of only spinning attempting to acquire the *run_lock*, we spin both attempting to acquire the lock and checking that the condition that required its acquisition is still true. If after a fixed number of iterations the thread neither acquire the *run_lock* nor determine that the thread no longer need to acquire it, thread yield the processor by depressing our priority. This optimization is called *Spin Polling*.

4.4. Continuation Recognition

The *default_filter* uses continuation recognition, an optimization first proposed and used in the Mach 3.0 kernel [7]. When the FILTER_INTERNAL_BUILD call of the *default_filter* prepares to call FILTER_INTERNAL_INVOKE of the thread it is activating, it checks to see if the *filter* of the resuming thread is the *default_filter*. If it is, then the *default_filter* can manipulate the *Internal Continuation* of the newly activated thread to be resumed directly, without needing to make the generalized FILTER_INTERNAL_INVOKE call.

Another form of continuation recognition occurs within *condition_signal*(). The continuation of a thread that is being continued by *Condition_signal* is *mutex_lock_solid*(). By examining the *Internal Continuation* and determining the actual mutex that the blocked thread will be attempting to acquire, *condition_signal*() can make a better choice about which queue to insert the thread into. If the mutex is currently held, then the thread is inserted into the mutex queue where it will later be resumed when the mutex is released. If the mutex is not held, then the thread is inserted into the *run_queue* with the expectation that it will succeed in acquiring the mutex when resumed.

5. Performance

This section describes two benchmarks designed to measure throughput and latency. We first compare the old and new implementations and then examine the effects of the previously described implementation decisions and optimizations. Two platforms are used to show these results: a Decstation 5000/20 Workstation with a 20MHz R3000 [11] and a Sequent Symmetry with twenty 20MHz 80386's [10].

5.1. Latency

To show the latency of blocking operations, we use a simple PingPong test that has two C-Threads context switch back and forth via a condition variable. Three versions of this program were created: one that sets the kernel thread limit to two threads, a second that sets the limit to one thread, and a third that does not limit the number of kernel threads, and *wires* each thread. Figure 5-1 shows the results of these tests.

	Decstation			Sequent		
Version	Old	New	Ratio	Old	New	Ratio
1 Thread	146μs	75μs	1.95	151μs	133μs	1.14
2 Threads	578μs	75μs	7.71	268μs	146μs	1.84
Wired	503μs	256μs	1.96	1225μs	447μs	2.74

Figure 5-1: PingPong benchmark results

Running the PingPong test with one kernel thread measures the latency of context switching. This is labeled "1 Thread" in Figure 5-1. As can be seen, nearly a two-fold improvement is seen on the Decstation. The latency on the Sequent improves also, but not by as much. This relative difference in improvement might be attributable to differences in the respective memory systems.

Running the PingPong test with the threads *wired* measures the cost of the synchronization primitive used by the underlying kernel. As described earlier, the old implementation uses *mach_msg*() for synchronization, while the new implementation uses thread priority depression. On the uniprocessor Decstation, the differences in synchronization costs result in nearly a factor of two performance improvement. The multiprocessor Sequent benefits more since explicit synchronization is not actually necessary to resume the second thread. With the old implementation, the second thread will not run until it receives a message from the first. This results in dead time during each iteration equivalent to the cost of a message send and message receive. Since the new implementation uses *Busy Waiting* in *wired* threads, the second thread will begin running immediately upon the condition being signaled. On a multiprocessor, the execution of the first thread's *thread_depress_abort*() overlaps the progress being made in the second thread. There is no equivalent cost to the message receive.

Running the PingPong test with two unwired kernel threads adds a way for progress to be made by allowing user-level context switching to occur. On the Decstation, a quantum, the amount of time before a pre-emptive kernel context switch will occur, is 15625µs. It is possible for a kernel thread to execute as many as 200 context switches before the another kernel thread will be pre-emptively scheduled. In the PingPong test, when the second kernel thread does run, if it cannot make progress, it yields the processor back to the first. If it can make progress, it will continue processing user-level context switches until another kernel context switch occurs. In contrast, the old implementation, with its higher latency and higher synchronization costs, can execute at most 100 user context switches between kernel context switches.

5.2. Contention Comparisons

To measure throughput, we ran a benchmark that generates lock contention and measures how long it takes multiple threads to make progress while contending for these locks. The contention benchmark creates a set of locks and associates a pool of threads which each lock. Each thread does some number of units of simulated work then acquires its lock, increments a common counter, and releases the lock. On the Sequent one unit of work takes 5.2µs.

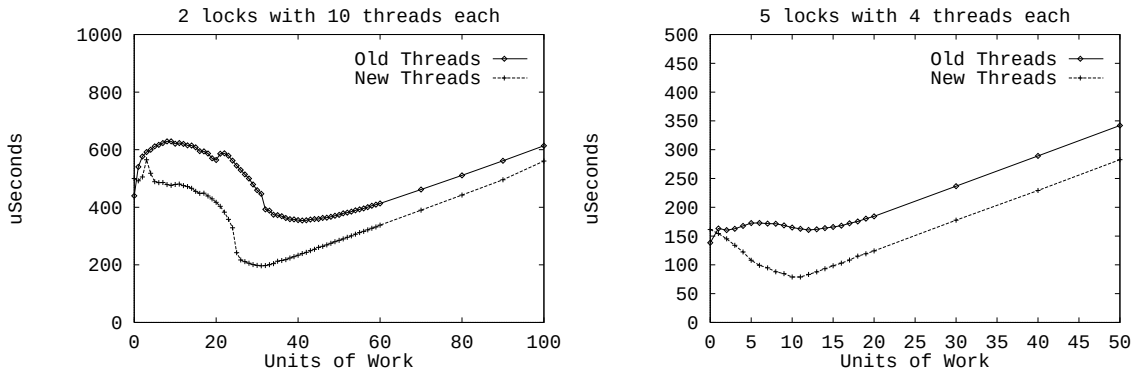

Figure 5-2: Comparisons of old and new thread packages on the Sequent

Figure 5-2 shows the contention benchmark on the Sequent comparing the old and new implementations with two different lock and thread parameters. If there were no lock contention, these graphs would be straight lines with slopes equal to the duration of one unit of work and intercepts equal to the overhead associated with creating and deleting the threads. As can be seen, the graphs are not straight lines and contention does exist as the amount of work done between acquisitions approaches zero.

The results of this benchmark show the throughput of new implementation is significantly greater than that of the old implementation. The overhead in the new implementation is lower because its primitives for creating and deleting threads are faster. In the old implementation there is a higher threshold of work above which contention is insignificant, which further widens the performance gap between the two implementations. When work approaches zero, the new and old implementations perform about equally well.

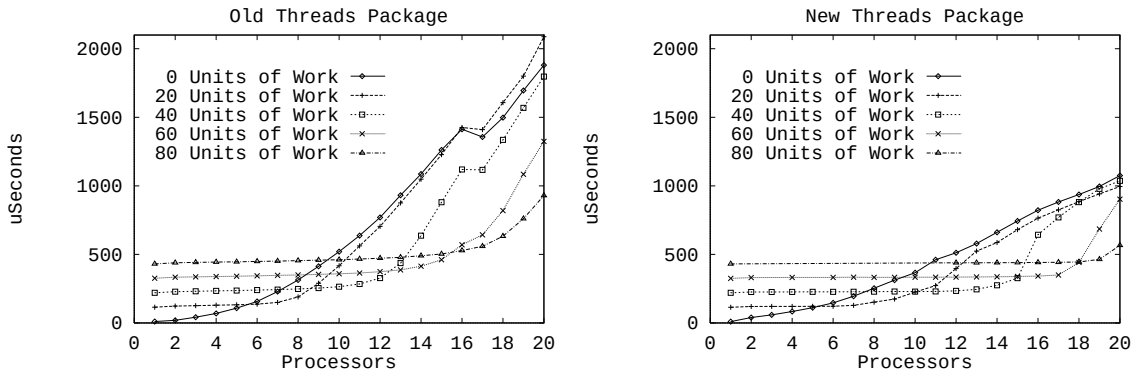

Figure 5-3: Old and New versions varying processors and amount of work on the Sequent

To better show the effect of adding processors to a finely grained application, we ran the contention benchmark with one lock and a varying number of threads, each doing a fixed amount of work. Figure 5-3 shows these results for a number of different amounts of work. As can be seen, the new implementation scales considerably better than the old one. If there is no contention, adding a thread that can run on an available processor should have no effect on the execution time of the benchmark. At some number of processors, the amount of time to execute this test begins increasing. With the old implementation and 40 units of work, contention begins

to affect the execution time at approximately 11 threads. The new implementation is not affected until approximately 13 threads. In general, it appears the new implementation can support two more threads without contention than the old one.

5.3. Effects of Different Techniques

To examine the effects of *Busy Spinning* and *Spin Polling*, three versions of the new C-Threads implementation were built with one or both of these features disabled. Figure 5-4 shows the contention benchmark run on the Sequent with 40 threads contending for one lock.

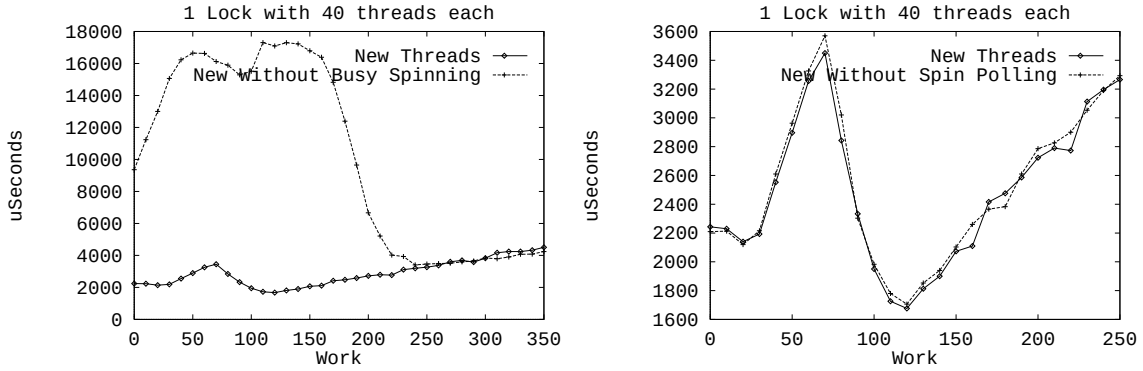

Figure 5-4: Effects of *Busy Spinning* and *Spin Polling* on throughput on the Sequent

The left graph in Figure 5-4 shows the effect of disabling *Busy Spinning*. The use of *Busy Spinning* provides an order of magnitude performance improvement in throughput under contention. The right graph shows the effect of disabling *Spin Polling*. While disabling *Spin Polling* alone does not make a significant difference, disabling both *Busy Spinning* and *Spin Polling* degrades performance by two orders of magnitude.

To better quantify the effects of these optimizations, measurements were made to examine the relative frequency of events, such as, acquiring the mutex while *Busy Spinning* or the *run_queue* becoming empty while attempting to acquire the *run_lock* in the idle thread. The contention benchmark was run. on the Sequent using one lock, 40 threads, and 80 units of work. As seen in Figure 5-4, 80 units of work is a point of prime contention. The total number of iterations for this test was 100,000 yielding 4,000,000 total attempts to acquire the mutex lock. Figure 5-5 shows these statistics.

	Count	Percent
Mutex Missed	3,913,671	97.8 of Total
Acquired while *Busy Spinning*	3,761,246	96.1 of misses
Acquired while *Spin Polling*	28,007	0.7 of misses
Mutex Blocked	124,418	3.2 of misses
Threads Idled	98,754	79.3 of blocked
C-Thread lost while *Spin Polling*	33,223	25.2 of waiters

Figure 5-5: Statistics for the Sequent with 1 lock and 40 threads

As expected from Figure 5-4, Figure 5-5 shows high contention on the mutex with a 97.8% chance of failing to acquire the mutex on the first attempt with 80 units of work. *Busy Spinning*

accounts for 96.1% of the cases where the mutex was missed. This is consistent with the effects shown in the first graph in Figure 5-4; removing *Busy Spinning* decreases performance by almost an order of magnitude. *Spin Polling* is only beneficial in about 1% of the misses but this translates to 18.4% of the cases where the thread was preparing to block. The remaining 3.2% of the misses result in the thread actually blocking and calling *cthread_block*(). Of the calls made to *cthread_block*(), almost 80% result in the kernel thread becoming idle. *Spin Polling* detects the emptying of the *run_queue* before acquiring the *run_lock* in 25% of the idle threads.

These results show that *Busy Spinning* is extremely important when there is contention on a mutex in order to avoid passing that contention to the *run_lock*. When contention is passed on to the *run_lock*, *Spin Polling* removes the need to acquire the lock in about 1 in 5 cases.

6. Conclusions

We have built a C-Threads library with lower latency and faster throughput than our previous implementation. Continuations decreased critical section size and made a number of important performance optimizations possible. The new C-Threads library allows applications to be built that will run well on both uniprocessors and multiprocessors without need to recompile or relink. In addition, the use of continuations, which allows the removal of the locking hierarchy and the state machine, both make the system faster and greatly simplifies the code.

7. Acknowledgments

Thanks go to everyone who helped in the work and commented on the paper. These include Brian Bershad, Rich Draves, David Golub, Joanne Karohl, Daniel Stodolsky, and Peter Stout.

References

[1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[2] Anderson, T.E., Bershad, B.N., Lazowska E. D. and Levy, H.M.
Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.
In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. October, 1991.

[3] Barton-Davis, P., McNamee, D., Vaswani, R. and Lazowska, E.D.
Adding Scheduler Activations to Mach 3.0.
In *Proceesdings of the 3rd Usenix MACH Symposium*. April, 1993.
To appear.

[4] Bershad, B.N., Redell, D., and Ellis, J.
Mutual Exclusion for Uniprocessors.
In *Proceeedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. October, 1992.

[5] Black, D.L.
Scheduling and Resource Management Techniques for Multiprocessors.
PhD thesis, Department of Computer Science, Carnegie Mellon University, July, 1990.

[6] Cooper, E.C. and Draves, R.P.
C Threads.
Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, February, 1988.

[7] Draves, R.P., Bershad, B., Rashid, R.F., and Dean, R.W.
Using Continuations to Implement Thread Management and Communication in Operating Systems.
In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. 1991.

[8] Golub, D., Dean, R.W., Forin, A., and Rashid, R.F.
Unix as an Application Program.
In *Proceedings of Summer 1990 USENIX Conference*. June, 1990.

[9] Goodman, J.R. and Woest, P.
The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor.
In *Proceedings of the 15th Annual Symposium on Computer Architecture*, pages 422--431. Honolulu, Hawaii, June, 1988.

[10] Intel.
386 Programmer's Reference Manual.
Intel, Mt. Prospect, IL, 1990.

[11] Gerry Kane.
MIPS RISC Architecture.
Prentice-Hall, Englewood Cliffs, NJ, 1988.

[12] Karlin, A.R., Li, K., Manassee, M.S., Owicki, S.
Empirical Studies of Competitive Spinning for Shared-Memory Multiprocessors.
In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. 1991.

[13] Leslie Lamport.
How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.
IEEE Transactions on Computers C-28(9):241-248, September, 1979.

[14] Robert Milne and Christopher Strachey.
A Theory of Programming Language Semantics.
Halsted Press, New York, 1976.

Table of Contents

1. Introduction
2. C-Thread Internals
- **2.1. Interface**
  - **2.1.1. Threads**
  - **2.1.2. Mutexes**
  - **2.1.3. Condition Variables**
  - **2.1.4. Kernel Threads**
  - **2.1.5. Thread Wiring**
- **2.2. Old Implementations**
  - **2.2.1. Locking**
  - **2.2.2. Blocking and Resuming**
  - **2.2.3. The State Machine**
- **2.3. Evaluation of Old Implementation**

3. Continuations
- **3.1. Building and Invoking Continuations**
- **3.2. The *filter* Mechanism**
- **3.3. Example of Continuation Use**

4. Optimizations
- **4.1. Locking Techniques**
- **4.2. *Busy Spinning* and *Busy Waiting***
- **4.3. *Spin Polling***
- **4.4. Continuation Recognition**

5. Performance
- **5.1. Latency**
- **5.2. Contention Comparisons**
- **5.3. Effects of Different Techniques**

6. Conclusions
7. Acknowledgments

List of Figures

Figure 3-1: Example of user blocking with *condition_wait*()
Figure 5-1: PingPong benchmark results
Figure 5-2: Comparisons of old and new thread packages on the Sequent
Figure 5-3: Old and New versions varying processors and amount of work on the Sequent
Figure 5-4: Effects of *Busy Spinning* and *Spin Polling* on throughput on the Sequent
Figure 5-5: Statistics for the Sequent with 1 lock and 40 threads