
Signals, timers, and continuations for multithreaded user-level protocols



Juan Carlos Gomez¹, Jorge R. Ramos² and Vernon Rego^{2,*},[†]

¹*IBM Research, 650 Harry Road, San Jose, CA 95119, U.S.A.*

²*Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, U.S.A.*

SUMMARY

Precise timing and asynchronous I/O are appealing features for many applications. Unix kernels provide such features on a per-process basis, using signals to communicate asynchronous events to applications. Per-process signals and timers are grossly inadequate for complex multithreaded applications that require per-thread signals and timers that operate at finer granularity. To respond to this need, we present a scheme that integrates asynchronous (Unix) signals with user-level threads, using the ARIADNE system as a platform. This is done with a view towards support for portable, multithreaded, and multiprotocol distributed applications, namely the CLAM (connectionless, lightweight, and multiway) communications library. In the same context, we propose the use of *continuations* as an efficient mechanism for reducing thread context-switching and busy-wait overheads in multithreaded protocols. Our proposal for integrating timers and signal-handling mechanisms not only solves problems related to race conditions, but also offers an efficient and flexible interface for timing and signalling threads. Copyright © 2006 John Wiley & Sons, Ltd.

KEY WORDS: timers; continuations; multithreading; parallel protocols; user-level protocols; asynchronous

INTRODUCTION

Precise timing mechanisms and asynchronous I/O are invaluable features for complex software applications. In the implementation of user-level protocols, for example, accurate timing and asynchronous I/O help enhance performance by increasing bandwidth utilization and reducing network-input polling overheads [1,2]. Most Unix kernels provide such features on a per-process basis, using signals to inform applications of asynchronous events. The interoperability of user-level

*Correspondence to: Vernon Rego, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, U.S.A.

[†]E-mail: rego@cs.purdue.edu

Contract/grant sponsor: DoD; contract/grant numbers: DAAG55-98-1-0246 and PRF-6903235

threads and process-level signals, however, is weak at best. Per-process signals and timers are grossly inadequate for complex multithreaded applications that require signals and timers that operate efficiently at thread granularity.

With a view toward efficient thread operations, we present a scheme that integrates asynchronous (Unix) signals and user-level threads; we describe a portable and efficient integration of this scheme into the ARIADNE system [3]. We make this presentation in the context of support for portable, multithreaded, and multiprotocol distributed applications, namely the CLAM (connectionless, lightweight, and multiway) communications library [2,4]. In the same framework, we propose the use of *continuations* as an efficient mechanism for reducing thread context-switch and busy-wait overheads in multithreaded[‡] protocols [5,6]. Our proposal for integrating timers and signal-handling mechanisms not only solves problems related to race conditions, but also offers an efficient and flexible interface for timing and signalling threads.

Timers are essential to a variety of complex software applications and, in particular, are critical in implementing communication protocols and real-time applications [7–12]. In sender-initiated protocols, timers are used to schedule packet retransmissions; in receiver-initiated protocols, timers are used to schedule requests for missing packets; in protocols with credit-based flow-control, timers are used to schedule transmissions of window-probe packets; in reliable multicast protocols [13], timers are used to schedule the periodic transmission of heartbeat packets. Polling-based protocol implementations and rate-based flow control schemes [14] also employ timers for scheduling message retrieval and message transmission, respectively. In general, timers lie at the heart of failure recovery schemes and are fundamental to mechanisms that schedule access to limited resources, e.g., the coordination of periodic control transfers (i.e. time-slicing) between distinct threads or processes.

The significance of a precise timing capability in the performance of user-level protocols is well-accepted[§], and issues of timing have been raised and addressed by researchers in the implementation of reliable uni- and multi-cast user-space protocols [15,16]. Timers are key to good performance and fairness in scheduling [2], as witnessed by our experiences with the CLAM communications environment, which offers distributed threads and multiprotocols in a single address space. CLAM demands timers that are portable, flexible, precise, and efficient. Timers must be portable and flexible to facilitate rapid ports and coding of protocol libraries and user applications. Timers must be precise to support efficient implementations of protocols for high-speed networks and real-time applications. Finally, timers must be efficient because they are used intensively by protocol modules.

The implementation of a portable, flexible, precise, and efficient timer subsystem and its integration with a user-level threads system depends on the solution to a specific problem, namely, the problem of integrating asynchronous per-process signals and user-level threads. Apart from providing asynchronous timing information, signals can also be used for asynchronous network input and for any other type of asynchronous communication initiated by the kernel. This signal-handling functionality can be used to advantage in many applications, including multithreaded user-level

[‡]Protocols implemented with multithreaded code are most commonly viewed as parallelizing protocols. We use the phrase *multithreaded protocols*, because multithreaded implementations of protocols are advantageous even on uniprocessor architectures.

[§]It must be noted that precise timing is most relevant in reliable communications that involve latencies of the order of tens of milliseconds or lower. Many popular uses of TCP, such as iSCSI, involve such typical latencies.

protocols, where asynchronous network input is a key factor in the delivery of low-latency messages. User-level multithreaded applications require flexible per-thread signal handlers that are capable of executing as threads. Most operating systems and user-level threads libraries, however, either lack or only provide some restricted subset of this important functionality. One of the main hurdles to be overcome in the provision of such flexibility is the efficient delivery of asynchronous signals and coordination with threads that are currently executing in critical sections. We focus on such a coordination, and present an interface for precise timing and rapid signal handling. We have found these features, implemented in ARIADNE, to significantly improve the performance of CLAM's (user-level) multithreaded protocols.

This paper presents two extensions to the ARIADNE threads system; these are intended to enhance efficiency and flexibility in the context of multithreaded user-level protocols. The first extension consists of a cost-effective solution to the challenge of integrating asynchronous signals and user-level threads. We present this enhancement to ARIADNE alongside a flexible and extended interface for handling asynchronous timed events and signal-activated threads. The second extension addresses various uses and possible implementations of continuations—a mechanism meant to improve the efficiency of multithreaded applications and threads packages; the interface to these extensions is described in the following. We finally present the results of a set of experiments that were conducted to evaluate the precision of timed events in the ARIADNE system. The key contributions of this paper include: an efficient, pragmatic scheme to integrate asynchronous events in user-level multithreaded environments; an efficient, precise and scalable timing facility for multithreaded applications (i.e. communication protocols); and, finally, the use of continuations to improve efficiency of multithreaded applications.

TIMERS FOR MULTITHREADED ENVIRONMENTS

To enhance software portability, ARIADNE's timers are based on the well-known Unix signals SIGALRM and SIGVTALRM. In principle, it is appropriate to multiplex Unix timing signals for delivery to a process, to enable different functions to execute at scheduled times. Such functions must be simple and run quickly, without accessing shared data structures and invoking thread primitives or *async-unsafe* functions[¶]. However, since such functions are of little use in practice, this method is not practical. To be useful, functions must do useful work and lend themselves to smooth and asynchronous interoperability with ongoing computations. Timed functions must be able to freely invoke thread primitives, yield control to new threads (i.e. effect a context-switch), and in general, invoke *async-unsafe* functions without restriction.

Consider an example of the kind of flexibility that is required from timed functions in building a real-time protocol using threads. A sending host may schedule timed functions to run and deallocate messages when their transmission deadlines have expired. These functions then return the buffer space that was occupied by these messages to a shared pool of buffers for reuse by 'transmitter' threads. As distinct active entities access the shared pool in an asynchronous manner, a thread lock must be

[¶]Async-unsafe functions are non-re-entrant functions that involve internal state stored in global variables.

used to guard against uncontrolled updates and, thus, maintain consistency of buffer state. Hence, timed functions must be able to exploit thread locks.

As another example of timed function requirements in protocols built with threads, consider the implementation of a reliable protocol. Here, message retransmissions are deftly handled by timed functions which are capable of invoking thread operations that may potentially block. The block may occur because a thread involved in a retransmission needs access to a shared data structure such as a connection record, or because of congestion- and/or flow-control mechanisms.

To respond to requirements such as those described in the above-mentioned examples, a threads system must offer an efficient form of protection against signals [17]. More specifically, the threads system should ensure that when a signal is delivered to a host process, its current thread^{||} must not be running within a critical section or, in general, within any async-unsafe function. Under such a guarantee, signal handlers (i.e. timed functions) are free to invoke any thread primitive or async-unsafe function. This freedom significantly enhances a thread system's flexibility and its ability to respond in a timely manner. The requirement is stringent: observe that if a signal handler is allowed to initiate a thread reschedule operation, all invocations of async-unsafe functions must be protected, even if these do not originate at the signal handler itself. Without this protection, a new thread that is scheduled by the signal handler may potentially invoke an async-unsafe function which is already being executed by the interrupted thread or some blocked thread.

Signal masks

The simplest way to ensure the required protection is to mask signals while the threads system is in a critical section or in some async-unsafe function. Such a solution, however, comes at a high cost because it involves interaction with the system kernel through system calls. Since most Unix system calls are async-unsafe, their invocation would have to be shielded by explicit signal masking operations, which, in turn, add significant overhead. Further, since the reschedule function itself is typically a critical section in a threads system, thread rescheduling operations would also need to be shielded by explicit signal masking. As a result, the cost of context-switching between user-level threads would climb to a value that is comparable to the cost of context-switching between kernel-level threads, and this cost increase would come without the advantages of kernel-level threads [18]. Due to the apparent high cost, we chose not to pursue a solution based on signal masks.

Global flags and timed events

To ensure the necessary protection against signals, an alternative is to exploit *global flags*. Through a global flag, a signal handler can be informed if the thread it interrupted was or was not executing within *unsafe code*** . If a signal handler determines that the interrupted thread was indeed executing in an unsafe region, it may either delay the execution of its own unsafe code until the thread exits the unsafe region, or it may defer its own execution until the interrupted thread voluntarily yields control. We call the first solution an *active-delay* approach and the second solution a *passive-delay*

^{||} Only the current or running thread(s) may be within an async-unsafe function at any given time.

** Henceforth, we use the term *unsafe code* to identify async-unsafe functions or critical sections.

(i.e. delay until the next context-switch) approach. Although the *passive-delay* variant makes the timing of timed functions less precise, it is cost-effective in that it sacrifices some accuracy in favor of efficiency. While postponing the execution of an asynchronous timed function until the next context-switch introduces a random delay, it also eliminates an overhead incurred in the *active-delay* scheme, namely, a repeated scan for pending timed functions each time the threads system exits a critical section.

A basic comparison

Motivated by our leading design considerations of portability, efficiency, and flexibility, we built a *timed-function* or *timed-event* subsystem based on global flags and integrated this subsystem with the ARIADNE threads system. The subsystem provides for both *active-delays* and *passive-delays*, with the latter being the default. In the passive-delay method, if timed-events cannot occur (i.e. timed-functions cannot be made to run) immediately, the Unix timer on which timing is based is loaded with a short time-to-expiry; this shortens a potentially large delay because the event is made to occur during the next thread context switch or when the timer expires, whichever occurs first. This small enhancement improves execution time accuracy by reducing the effects of Unix's coarse-grained process-oriented timers in applications with high context switching rates. To determine whether events have expired at a finer granularity than that provided by the Unix per-process timer, timers must be read; hence, the increase in accuracy comes at the cost of timer handling overhead.

To compare the performance of the global flag and the signal-mask methods for handling signals, we made some simple measurements on the costs of system calls used in their implementations. The cost of reloading a Unix timer on a 70 MHz Sun Sparcstation-5, using the `setitimer()` system call required in the global flag implementation, is approximately $29 \mu\text{s}^{\dagger\dagger}$. The corresponding cost of the `sigprocmask()` system call required in the signal-mask implementation is approximately $27 \mu\text{s}$. Considering the rate at which each of these system calls will be made for each method, the global flag implementation shows clear advantages. With the signal-mask method, the `sigprocmask()` system call must be used to shield most of ARIADNE's thread primitives—in particular, its rescheduling routine—from signals delivered while they are being run. Further, this system call will be invoked frequently, independently of the number of events scheduled. In contrast, the expense of an extra timer reload with `setitimer()` for the global flag method is incurred only when signal delivery encounters the threads system in a critical section. Given that the likelihood of this event is small relative to potential `sigprocmask()` invocations, the cost of the global flag method is bound to be smaller.

Implementation issues

The timing and signal handling functionality that supports the CLAM multiprotocol library is implemented as a module in ARIADNE [3,19]; the structure is shown in Figure 1. The tight integration of this module with the rest of the threads system is pivotal in ARIADNE's ability to be flexible and

^{††} Although `setitimer()` is used by the two methods at scheduling time, the global flag scheme may incur the cost of extra calls to this function when events cannot be run at expiration time.

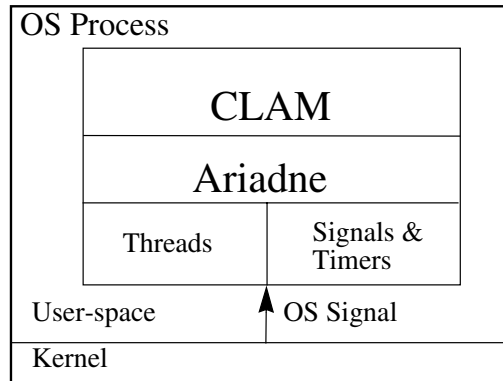


Figure 1. Timers and signals in ARIADNE.

deliver precise and efficient timing and signal handling. This design also enables users to develop timer- and signal-activated routines that are free to invoke thread primitives. Furthermore, the design facilitates low-cost and precisely synchronized timer- and signal-activated executions in concert with other threads. Although a timing and signal-handling capability can be implemented outside the threads library, the cost of crossing an additional interface makes this scheme unattractive.

To enhance portability, we have implemented ARIADNE's timers using the well-known Unix signals `SIGALRM` and `SIGVTALRM`. These signals can be used for sharing CPU time between different runnable threads (i.e. time-slicing) and for the timed-bound scheduling of functions and threads. ARIADNE's timers effectively multiplex process-level timing signals to obtain an efficient thread-level timing facility. ARIADNE's embedded signal handling code captures and smoothly integrates Unix's implicit signal asynchrony with the threads system, in a manner that promotes timing efficiency and prevents race conditions.

In ARIADNE, each event that is scheduled to run at a given time is represented by a function pointer, a parameter, a flag (specifying its type), and its expiration time. Timed events are stored in a heap data structure (i.e. a delta queue) until their timers expire. When an event's timer expires, its associated function is invoked either as a thread or as a function, depending on the flag held in its event record. The number of event heaps held by a process may be as high as the number of its process timers. Each heap is bound to and timed by a Unix timer. Event handlers that execute as functions are very efficient, but are restricted in the sense that they cannot invoke thread primitives; such an invocation can lead to awkward results since ARIADNE currently does not support it. For example, consider a thread that waits on a semaphore and is just about to yield control to another thread. Further, assume that the waiting thread is interrupted by a timed event that must run as a function and wait at the same semaphore at which the interrupted thread waits. Since the event is run on the stack of the waiting, interrupted thread, it is possible for a single thread to end up simultaneously queued more than once at the same semaphore. ARIADNE currently does not support this because of extra handling overheads. In contrast, however, event handlers may execute as independent threads with unrestricted access to thread primitives.

This gives considerable flexibility at the expense of scheduling overhead that is a little higher than function invocation.

The *Optimistic Event Handler* (OEH) model is yet another execution model for ARIADNE's timed events. This model is based on the concept of Optimistic Active Messages (OAM) [20]. Here, the intention is to free the user from having to decide which events should be handled as functions and which events should be handled as threads. OEHs execute as functions until the point at which they cause a reschedule. At this point, a new thread is automatically created to resume the rest of the rescheduled event's computation. Such handlers tend to reduce the scheduling overheads of events that invoke threads primitives but seldom block or cause rescheduling operations; naturally, it is wasteful to create new threads for such events in general.

ARIADNE's current implementation of timed events may be described as follows. Each Unix timer is linked to a heap which contains events with times that await expiry. When an event is to be scheduled, or when its time expires, the Unix timer with which it is linked is loaded with the expiration time of the event, from the same heap, whose expiry time is next. Timers are switched off when there are no pending events.

All timing signals are processed by a single handler. First, the signal handler determines whether the threads system is currently executing unsafe code, i.e. whether the global flag that identifies this condition is set. If indeed this is the case, the processing of an event whose time has just expired is postponed; the actual delay depends on the variant of the global mask solution being employed. If the threads system is not in a critical section when an event's time expires, the event is processed immediately in a manner that depends on whether it is a function, thread, or optimistic handler.

ARIADNE supports the following types of timed events: (thread) time-slicing events, thread-resumption events, functions, thread-bound delay-sensitive events, and thread-bound delay-tolerant events. A (thread) time-slicing event causes the preemption of the currently running thread and a reschedule operation which runs the next runnable thread with highest priority. If a runnable thread with priority equal to or higher than the preempted thread does not exist, the preempted thread is allowed to resume. A thread-resumption event causes the preemption of the currently running thread, insertion of a previously suspended thread back into the ready queue, and a reschedule operation; the reschedule causes the highest priority runnable thread to run. When a function's timer expires, it runs on the stack of the currently executing thread.

A thread-bound event is a function that must execute as a thread but does not have its own stack. This class of events is specifically designed to support the scheduling of functions that may invoke thread primitives but run to completion quickly; these do not justify the overhead of thread creation. The ARIADNE threads system maintains an internal linked-list of idle threads that are used exclusively for the purpose of executing thread-bound events. New threads are added dynamically to this list in order to meet increasing concurrency levels as required by the application. When a thread-bound event's timer expires, a thread is extracted from this linked-list to serve as its host. When the event's execution is complete, the thread is returned to the linked-list of idle threads where it is free to be used by new events. A set of thread-bound events without strict deadlines can be hosted by a single thread if these events expire at about the same time. Such support enables many thread-bound events to run almost simultaneously, without the additional overhead of scheduling distinct threads for each event. There is a limit on the number of such events that may be hosted by a single thread; this helps prevent unbounded delays in event executions. Thread-bound events with strict deadlines are assigned to individual threads, thus ensuring that they will run without unpredictable delays.

Although ARIADNE's timed events offer efficiency and flexibility in scheduling, their precision depends on the accuracy of the underlying Unix process-timers. In our current implementation, we reduce this dependency by scanning for expired events after every thread context-switch. Even though this technique generally improves timing precision in applications with high rates of thread context-switching, it does little to improve the precision of timing in applications that exhibit low context-switching rates. Fortunately, applications such as CLAM use threads intensively with a high rate of thread context-switching, and thus witness a precise timing capability.

In addition to the precision limitations addressed above, the efficiency of ARIADNE's timers falls when the number of pending events in the event queue is large. This is a direct result of the $O(n)$ cost of event insertion in the delta-queue data structure used in our implementation. Although a heap would provide an $O(\log(n))$ cost in insertion time, a delta-queue is simple, and its insertion time can be controlled if a pattern is seen in the way timers are used. Further, a doubly-linked delta-queue offers a constant cost with deletion, compared to the $O(\log(n))$ cost of deletion with a heap.

CLAM employs two techniques to achieve efficient timing with a doubly-linked delta queue. First, it minimizes the number of waiting events by controlling the number of events that can be scheduled simultaneously. Second, it exploits information on the expiration time of events that are being scheduled, to enable it to perform efficient insertions in the delta-queue. CLAM uses ARIADNE's timers only for events that require precise timing and occur simultaneously in large numbers. To reduce insertion time in the delta-queue we ascertain whether the sequence of expiration times of a set of events currently being scheduled is an increasing or a decreasing sequence. If the sequence is increasing, the set's insertion point is determined by searching the queue starting at its tail (i.e. the queue element with the largest expiration time). If the sequence is decreasing the search starts at the head of the queue (i.e. the queue element with the smallest expiration time).

The scheduling of events that require less precise timing and which may potentially occur simultaneously in large numbers is handled at the CLAM library level where they are hosted by a single timed thread. This enables a reduction in the number of pending events in ARIADNE's delta queue, so as to ensure a reasonable precision and efficiency in timing. The suspension period of the timed thread that manages these CLAM library-level events is set to match the minimum timing granularity required by the events it hosts. In this implementation, most of the operations performed by the threads system that involve timers are low-cost operations. A scan for expired events after a thread's context-switch incurs only a small constant cost; this operation does not require a function call unless an event has actually expired. A reload of Unix timers also incurs only a small constant cost. The cost of event insertion in a delta-queue is a small constant when event expiration-time sequences are increasing or decreasing. In total, all of these factors combine to yield an excellent overall performance, precision, and functionality.

SIGNAL-BASED THREAD ACTIVATION

The ARIADNE threads library offers another important feature, namely, signal-based thread activation. This functionality is crucial in the provision of efficient support for the implementation of multithreaded communication protocols. Our signal-handling primitives are especially tailored to the implementation of multiple user-level protocols within a single address space. ARIADNE provides an

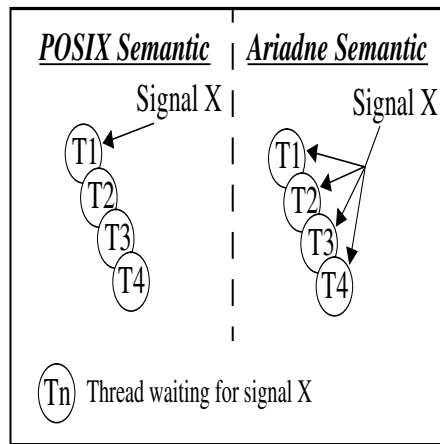


Figure 2. Signal wait semantic: POSIX versus ARIADNE.

interface that is similar to the signal-wait feature offered by POSIX threads, but with a few semantic differences. Using this interface, ARIADNE threads may wait for specific signals and be promptly activated when these signals are delivered. As an illustration, consider the following uses: a thread that is activated by a SIGCHLD signal offers an efficient and clean mechanism for capturing state changes in child processes in a multithreaded application; similarly, a SIGIO-activated receive thread offers an efficient, low-latency mechanism for receiving messages that arrive at unpredictable times.

As indicated earlier, the handling of signals in multithreaded systems is a complicated task because of potential race conditions. Although it is possible to resort to simple solutions that prevent race conditions, these may involve significant overheads. In our current implementation of signal-based thread activations, we trade efficiency against precision in signal delivery to obtain a design that is similar to our design of timed event support. ARIADNE's signal-wait functionality differs from the functionality provided by POSIX threads in that the occurrence of a signal awakens all waiting threads instead of a single thread, as shown in Figure 2. This semantic was implemented for a special purpose—to respond to the need for asynchronous thread activation in multiprotocol environments. Here, multiple receive threads are very useful in that they can exploit the concurrency inherent to multiple protocols that run within one address space.

To demonstrate how the above functionality can be used in multiprotocol environments, consider a situation in which multiple receive threads are used—one for each protocol—with each thread operating on a distinct socket. Further, assume that each socket has a SIGIO signal that is enabled. In the best scenario, the only receive threads to awaken are those that handle protocols for which messages actually arrive. With the POSIX semantic, only one thread is allowed to resume, and thus there is no way to ensure that this thread corresponds to the protocol(s) for which messages have arrived. Unfortunately, Unix signals are of no help in obtaining such selectivity since they fail to provide the information that would be necessary to differentiate between signals generated by different sockets.

Therefore, the only remaining alternative is to awaken all the sleeping receive threads and to let them determine which of them have actually received messages. The POSIX signal-wait semantic does not permit a direct implementation of this solution because only one thread can be awoken with a given signal. The ARIADNE semantic, however, allows all of the threads waiting on a given signal to resume execution upon the delivery of such a signal. Through the use of non-blocking receive calls, each thread can then determine whether indeed it is supposed to retrieve the input that generated the signal. While this solution is effective in terms of functionality, it necessarily entails scheduling overhead due to the activation of receive threads related to protocols with potentially no incoming messages. We feel, however, that this is a reasonable and portable mechanism for using a multiplexed SIGIO signal, especially when the number of simultaneously running protocols is not large.

CONTINUATIONS FOR MULTITHREADED PROTOCOLS

Continuations are useful abstractions that can be exploited to reduce memory and context-switching overheads in multithreaded environments. Through their use, a thread can discard its low-level execution state (i.e. its stack) and block, while preserving only crucial state information within a continuation record. In general, this abstraction uses little memory and can be manipulated by the application itself, to yield enhanced efficiency and flexibility. A continuation can be implemented as a function (with a set of parameters) that runs just like a thread except at the time of a context-switch. When the threads system effects a reschedule operation to yield control to a continuation, the system does not need to run the usual save and restore procedures that are required in switching control between two threads. When a continuation runs, it executes on the stack of the thread that yielded control to it, thus eliminating the overhead of a (thread) context-save and a context-restore operation.

Continuations have previously been studied in the context of kernel implementations [21]. They have also been used in the framework of user-level threads libraries for multiprocessors [22]; the intent was to reduce locking effects resulting from the implementation of register save and restore operations within critical sections of the threads system. We have no knowledge, however, of any attempt for exploiting continuations for the support of multithreaded user-level protocols. Here, we focus on the application of continuations to such protocols.

An appealing use of a continuation involves its ability to store the execution context of only a part of a thread's code. That is, a continuation may run only that part of code which causes a thread to potentially block or wait on a busy-cycle. In this way, the remaining thread code is free to continue its execution while the continuation independently waits for the blocking condition to clear. This particular use of continuations enhances concurrency while simultaneously eliminating context-switching overheads that can otherwise be incurred when a thread blocks. Similarly, continuations reduce the overheads of unnecessary busy-waits on locks and enable execution progress by allowing the remaining thread code to run; all of this comes without the cost of additional reschedule operations. Although it may be tempting to use continuations in all of the potentially blocking sections of a thread's code, this is workable only when the order in which a thread's continuations are executed does not conflict with the remaining thread code. Despite this limitation, the occurrence of thread blocks and busy-waits in multiprotocol implementations are so frequent that continuations offer the potential for high savings.

Consider an illustration of how continuations can help efficiency in multithreaded protocol implementations. In buffer handling, when a thread completes its usage of a message buffer it typically

recycles the buffer for reuse by returning it to a shared pool. This shared pool of buffers is protected by a lock or a semaphore to keep its state consistent. While it is crucial for a thread that requests a buffer to actually obtain the buffer before continuing with its execution, it is not crucial for a thread that is releasing a buffer to actually wait until the buffer is placed in the shared pool. After initiating a buffer release, a thread may continue with other tasks even before the buffer is returned to the shared pool. This situation offers an ideal opportunity for a continuation-based implementation of buffer release; performance improves because it circumvents unnecessary thread blocks or busy-waits^{‡‡}. To implement this, whenever a current thread initiates a buffer release and finds that access to the shared pool has been locked by another thread, it immediately posts a continuation to complete its task. The continuation will run at a future time, on the stack of a thread that gives it control; it is appropriate to implement this continuation as a function that runs to acquire the lock, quickly returns the buffer to the shared pool, releases the lock, and then terminates.

Another example of the use of continuations can be found in implementations of reliable protocols built around threads. For instance, a receive thread may post messages to a receive-queue that is concurrently accessed by several (application-level) threads. If access to the receive-queue is protected by a lock, its state can be kept consistent. However, if the receive thread spins on a busy-wait or blocks while another thread accesses the receive-queue, network utilization falls and context-switching costs rise. A solution based on continuations will work as follows. When the receive thread finds the receive-queue locked by another thread, it creates a continuation that will post its message when the lock is eventually released. In this way, the receive thread may process incoming messages rapidly, without having to block or spin in waiting to post messages to upper protocol layers. In this continuation-based solution, a receive thread will continue to run without interruption to service a burst of packet arrivals. This will be true even when the receive-queue is locked by a thread that is currently suspended. Thus, in addition to eliminating busy-waits and unnecessary context-switches, continuations also offer improved cache locality for a receive thread's code.

Although continuations offer a mechanism for performance enhancement, their use is limited to situations in which sections of a thread's code can run independently of other sections. Consider an example of a situation where this would not work. In implementing a sender-initiated reliable protocol module, a receive thread may need to update a connection record that is protected by a lock before sending an acknowledgment to a sending thread. If the connection record update is delayed, the acknowledgment sent by the receive thread may end up carrying obsolete or duplicate information; this leads to poor use of network bandwidth and equally poor synchronization between sender and receiver.

PROGRAMMING INTERFACE

In this section we present the programming interface to ARIADNE's timed events and signal-activated threads. The interface provides a user with primitives to schedule and cancel timed events in their different forms: these include timed functions, timed thread-resumptions, timed thread-bound

^{‡‡}Posting a continuation is a short, simple operation while releasing buffers in a complex system may even involve thread-rescheduling (i.e. if a high-priority thread is waiting for the buffer).

functions, and time-sliced events. This section also contains a description of mechanisms to set, clear, and activate ARIADNE's signal handlers, a primitive that enables threads to wait on specific signals, and calls that enable pre- and post-execution actions for specific threads. In terms of implementation, the pre-execution mechanism resembles signal delivery in Unix, but set in a user-level threads framework. Post-execution actions are operations of small duration, scheduled to occur immediately after a specific thread yields control. As this useful and flexible functionality comes without adverse affects on performance, we have incorporated it into the basic threads system.

Timed events

ARIADNE provides special support for two types of timed events: one type of event is used for time-slicing and the other is used for timed thread-resumptions. The primitives `a_defslice()`, `a_clearslice()`, and `a_initslice()` are used to handle time-slice events. They enable, disable, and initialize, respectively, the time-slicing of threads at a given priority level. When a thread is activated and its scheduling policy indicates that it is to run for a time-slice, an event is posted to flag the time-slice's expiry. This event occurs at the end of the thread's time quantum and causes a new thread—and possibly also a new time-slice event—to be scheduled. If a thread that is to run for a time-slice actually yields CPU control before its time quantum expires, its time-slice event is cancelled before control is transferred to another thread. The ARIADNE primitive `a_defslice()` defines the priority levels at which time-slicing is enabled and indicates the specific timer used. The `a_clearslice()` primitive is used to cancel a previously scheduled and pending time-slice event. The `a_initslice()` primitive is used to initialize time-slice events.

The prototypes of the calls used to handle time-slicing are

```
void a_initslice();
void a_clearslice();
void a_defslice( int t, int p, u_long s, u_long u );
```

The parameter `t` specifies the type of timer to be used in signalling time-slice expiry: possible values for this parameter include `REAL` (real timer) and `EXEC` (virtual timer). The parameter `p` can assume either a valid ARIADNE priority level or the constant value `ALL`. It defines priority levels at which a time-slicing mode of operation must be used in thread scheduling. The constant `ALL` indicates that this mode applies to all priority levels. The `s` and `u` inputs to `a_defslice()` specify the time-slice length (quantum) in seconds and microseconds, respectively.

The primitive `a_sleep()` is used to schedule timed thread-resumptions by suspending threads for specific periods of time. By invoking `a_sleep()`, the thread first creates an event record for itself and then schedules a wake-up event for this record. When the specified time elapses and the wake-up event occurs, the event action returns the thread to the ready queue and effects a reschedule operation. The call is flexible in that the application may select the particular timer that is used to signal the end of a thread's suspension interval. The prototype of the `a_sleep()` primitive is

```
void a_sleep( u_long s, u_long u, int t );
```

The parameters `s` and `u` specify the suspension time in seconds and microseconds, respectively. The parameter `t` specifies the timer to be used to signal the thread-resumption event; possible values for this parameter are `REAL` (real timer) and `EXEC` (virtual timer).

Timed thread-bound events and timed functions are manipulated with help from the `a_schevnt()` and `a_deschevnt()` primitives and the following set of C-language macros:

```
a_schevnt_r_t()
a_schevnt_r_vt()
a_schevnt_a_r_t()
a_schevnt_a_vt()
a_cnclevnt_r_t()
a_cnclevnt_vt()
```

The prototype of the `a_schevnt()` call is

```
void * a_schevnt( void (*fn)(void *), void * p, struct timeval *t, \
                 u_long fl );
```

This primitive schedules a function specified by the parameter `fn` to be invoked with an argument `p` at a specified time `t`. The particular timer (i.e. real or virtual) used by the event, the manner in which the time parameter `t` is interpreted (i.e. relative or absolute), and the manner in which the event is to be executed are all specified by the single parameter `fl`. This parameter is an ‘OR’ combination of the following flags: `ABSEVNT`, `RELEVNT`, `PRIEVNT`, `MPREVNT`, `RTQEVNT`, `VTQEVNT`, and if applicable, also the priority at which the event must execute; the priority value is located in the upper half of the `fl` parameter. The constants `ABSEVNT` and `RELEVNT` specify whether parameter `t` is an absolute time or a time relative to the current time, respectively. The `PRIEVNT` flag is set for delay-sensitive thread-bound events and cleared for timed functions and delay-tolerant thread-bound events. The `MPREVNT` flag is set for delay-tolerant thread-bound events and cleared for timed functions and delay-sensitive events. The `RTQEVNT` and `VTQEVNT` flags specify whether the event is to be scheduled in the real timer queue or in the virtual timer queue, respectively. A successful invocation of the `a_schevnt()` primitive returns a pointer to the event structure that is created.

By invocation of the `a_deschevnt()` primitive, the pointer returned by `a_schevnt()` may be used to cancel the event before its potential occurrence. The prototype of the function `a_deschevnt()` is

```
void * a_deschevnt( void * p );
```

When invoked, this function cancels a given event before it can occur. The parameter `p` is a pointer to the event structure returned by `a_schevnt()` when the event is scheduled. If the event is found to have already occurred, the function returns a null value. A non-zero return value indicates that the event was successfully cancelled before its occurrence.

The other primitives mentioned in relation to `a_schevnt()` and `a_deschevnt()` are C-language macros that play a supporting role. The `a_schevnt_r_t()` macro is defined by

```
#define a_schevnt_r_t( f, p, pr, t ) \
    a_schevnt( f, p, t, ((pr << 16) | PRIEVNT | RELEVNT | RTQEVNT) )
```

This macro schedules a delay-sensitive thread-bound event (`PRIEVNT`) with relative timing (`RELEVNT`). The scheduled event is timed by the real timer (`RTQEVNT`). The priority at which the

event's host thread must run is specified by parameter *pr*. This priority must be a valid integer priority for ARIADNE threads (i.e. an integer between 0 and 8).

The macro `a_schevnt_r_vt()` is defined by

```
#define a_schevnt_r_vt( f, p, pr, t ) \
    a_schevnt( f, p, t, ((pr << 16) | PRIEVNT | RELEVNT | VTQEVNT) )
```

This macro schedules a delay-sensitive thread-bound event (PRIEVNT) with relative timing (RELEVNT). This scheduled event is timed by the virtual timer (VTQEVNT) and must be hosted by a thread that runs at priority level *pr*.

The macro `a_schevnt_a_rt()` is defined by

```
#define a_schevnt_a_rt( f, p, pr, t ) \
    a_schevnt( f, p, t, ((pr << 16) | PRIEVNT | ABSEVNT | RTQEVNT) )
```

This macro schedules a delay-sensitive thread-bound event (PRIEVNT) with absolute timing (ABSEVNT). The scheduled event is timed by the real timer (RTQEVNT) and must be hosted by a thread that runs at priority level *pr*.

The macro `a_schevnt_a_vt()` is defined by

```
#define a_schevnt_a_vt( f, p, pr, t ) \
    a_schevnt( f, p, t, ((pr << 16) | PRIEVNT | ABSEVNT | VTQEVNT) )
```

This macro schedules a delay-sensitive thread-bound event (PRIEVNT) using absolute timing (ABSEVNT). This event is timed by the virtual timer (VTQEVNT) and must be hosted by a thread that runs at priority level *pr*.

An event that is pending may be identified and cancelled, based on its potential parameter value (i.e. the parameter *p* described previously). The macro `a_cnclevnt_rt()` is defined by

```
#define a_cnclevnt_rt( p ) exdp( RTEQ, p )
```

This macro examines events in the real-time queue and cancels the event whose parameter value is *p*. The macro `a_cnclevnt_vt()`, which is also used to cancel events, is defined by

```
#define a_cnclevnt_vt( p ) exdp( VTEQ, p )
```

This macro is similar to `a_cnclevnt_rt()` except that it examines the virtual time event queue instead, for event cancellation. If two or more events have the same parameter value, only the event with the earliest scheduled time of occurrence is cancelled. Both macros for event cancellation return a null value if the event to be cancelled is not found in the queue. If the cancellation is successful, a non-null pointer to the event record is returned.

Handling signals in ARIADNE

Although signals can be processed by application-level handlers, ARIADNE's global signal handlers may be installed for a subset of Unix signals. These handlers enable threads to wait for specific signals. ARIADNE's internal handlers are installed with the `a_setsig()` call and removed with the

`a_clearsig()` call. Once an internal handler for a specific signal has been installed, a thread may wait for the signal by invoking the `a_waitsig()` primitive. A thread may activate other threads—which block upon calling `a_waitsig()`—by invoking the `a_raisesig()` primitive. The latter call is equivalent to actually delivering a signal to a process, but without the expense of crossing a kernel boundary.

The prototypes for signal-related primitives in ARIADNE are

```
void a_setsig( int sig );
void a_clearsig( int sig );
void a_raisesig( int sig );
void a_waitsig( int sig );
```

The parameter `sig` defines the Unix signal for which the ARIADNE internal handler is being installed (`a_setsig()`) or removed (`a_clearsig()`). In the case of `a_raisesig()`, the `sig` parameter indicates the Unix signal being raised. In the case of the `a_waitsig()` call, the `sig` parameter indicates a Unix signal for which a thread must wait; only the SIGIO, SIGCLD, and SIGINT Unix signals are currently supported. Other signals may be readily incorporated, if necessary, through minor modifications to the threads library. The current implementation can support up to 32 distinct signal types without major modifications.

Pre- and post-execution actions

An ARIADNE thread may be pre-empted at any time by an event or a thread that has a higher priority. There are situations where it is useful for an application to be able to run a given piece of code immediately before a particular thread runs, or immediately after it either voluntarily yields control or is pre-empted. To provide this flexibility, ARIADNE places pointers to two functions and their parameters in every thread record. The default setting of these two pointers is null. Once a thread is created, the application can set the function pointers to point at specific code by invoking the `a_setschdf()` and `a_setdschdf()` primitives.

The `a_setschdf()` primitive specifies the thread for which pre-exec function is being set. This function is to run immediately before the thread runs, every time the thread is being given control. The pre-exec function runs on the stack of the thread that yields control to the thread for which the pre-execution function is set. The prototype for this primitive is

```
a_setschdf( (struct tca *)t, (void (*)(void *))f, (void *)p );
```

The parameters specify the thread `t` for which the pre-exec function is being set, a pointer `f` to the pre-exec function, and the parameter `p` to be passed to this function.

When invoked, the `a_setdschdf()` primitive sets a pointer to a function that is to run when a thread either yields control or is pre-empted. The prototype for this primitive is

```
a_setdschdf( (struct tca *)t, (void (*)(void *))f, (void *)p );
```

The parameter `t` specifies the thread for which the post-exec function is being set. The parameter `f` defines the function that is to run when the thread either yields control or is pre-empted.

The parameter p must be passed to the post-exec function at run time, when this function is entered. In contrast to the pre-exec function, the post-exec function runs on the stack of the thread for which the function was set.

The above functionality is used by the CLAM library to support time-sliced thread executions, along with deadline scheduling. For example, assume that a set of threads is currently sharing the CPU in a round-robin fashion, with each thread obtaining up to a given maximum-length slice of CPU time. In addition, also assume that the number of threads in the set may change dynamically, and one of these threads in particular—a receive thread—has a real-time execution deadline. That is, the receive thread must be rescheduled for execution within a given time interval whenever it either yields control or is pre-empted. As a thread may not consume its entire allocated time-slice before context-switching, and because the size of the set of threads changes dynamically, it is difficult to guarantee that a particular thread will run by a given deadline. One way of enabling a thread to meet a deadline is to break the static round-robin scheduling cycle and give control to the thread with the deadline constraint. The ARIADNE primitives described here enable a thread with a real-time deadline to schedule a high-priority event which, upon its occurrence, breaks the round-robin cycle of thread schedules and immediately yields control to the thread with a deadline. CLAM uses this idea in scheduling receive threads that are not interrupt-driven but that are scheduled to run in round-robin fashion with other threads. This approach enables CLAM to control message latencies and guarantee a reasonable level of communication service in polling-based protocols, including situations with a large number of threads.

EXPERIMENTAL RESULTS

We present the results of a set of experiments designed to compare the precision of ARIADNE's timers to the precision of regular Unix per-process timers. The comparison is done under different processor loads. All experiments were conducted on a SPARCstation 5 (70 MHz, 32 MB memory, SunOS Solaris 5.5.1). We measured the actual time interval defined by the timers using the `gethrtime()` call provided by SunOS Solaris. Processor loading by other applications was observed to be negligible at the time of the experiments. Unless stated otherwise, all processes were scheduled at the same priority level, namely, the default priority assigned by the operating system to regular non-privileged users. Processes were scheduled under the time-sharing scheduling class, unless stated otherwise.

Experiment 1: real-time timer precision

To compare timing precision, we ran two processes—one that measured ARIADNE's timing precision and another that measured Unix's timing precision. In each case timers were repeatedly set to expire at the end of a fixed-length time interval of 10 ms, i.e. this was the time to timer expiry requested. Due to the granularity with which the underlying system scans for timer expiry, however, the actual elapsed (and measured) time to timer expiry differs from the requested time. To measure this *timing offset** we computed and recorded 200 samples of the actual elapsed time to timer expiry. These sampled values

*From an application's perspective, this time is actually an error, since the system responds with a time that is different from the time the application requested.

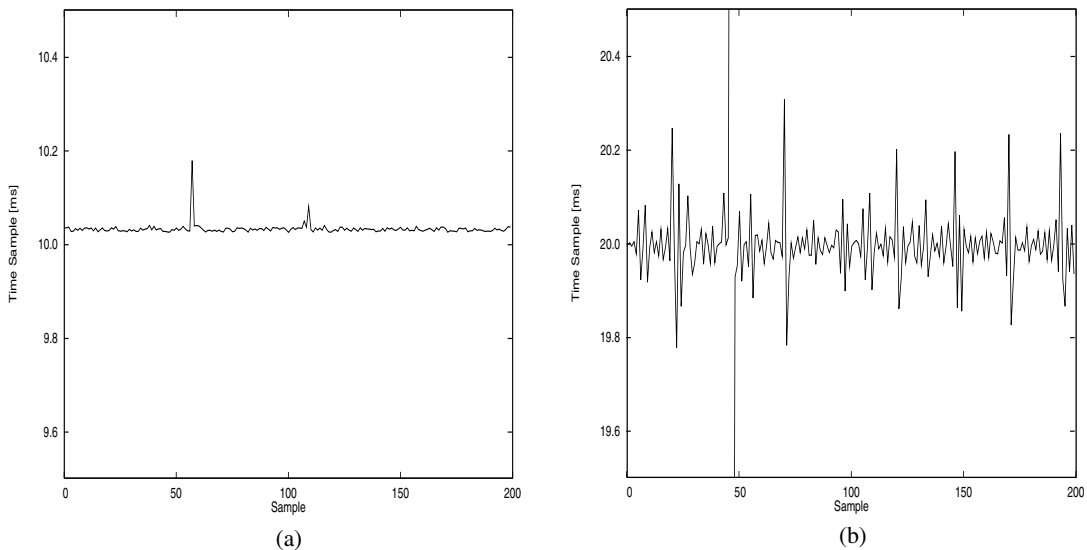


Figure 3. (a) ARIADNE versus (b) Unix timers: detailed view.

are displayed in Figure 3 for both ARIADNE and Unix. The ARIADNE process used in this measurement consists of a thread that invokes `a_sleep()` and measures its own actual suspension time using the `gethrtime()` call. The process employing Unix timers directly used the `setitimer()` and `pause()` system calls to suspend itself for a given time interval. This process also measured its actual suspension time using `gethrtime()`.

The results shown in Figure 3 emphasize two important points. First, measurements obtained from the Unix timer exhibit a large timing offset, which is in the order of 10 ms for most samples. This timing error is due to the clock-tick granularity that was configured for the workstation testbed. For this environment, the tick parameter was set to a frequency of 100 ticks per sec, or equivalently, an interval of 10 ms between consecutive ticks. The finer the tick granularity, the more precise will be the Unix timers. There is, however, a performance penalty that a Unix kernel must pay in exchange for a finer clock-tick granularity, i.e. the finer the tick granularity, the higher the corresponding process-timer management overhead. For this reason, it is common to configure the tick rate to be some reasonable level, usually in the order of milliseconds.

The second important detail worth noting in Figure 3 is the markedly high variability exhibited by the Unix timer measurements relative to the low-variance measurements given by ARIADNE's timers. We believe that this is due to blocking: the process that hosts the Unix timing test blocks in the kernel within the `pause()` system call, while the process that hosts the ARIADNE timing test simply continues to run some other ready thread during the suspension interval. The results shown here are consistent with measurements made for different timer values, as shown in Figure 4. In this figure the graphs corresponding to the different timer values are overlaid in a single frame, so that they

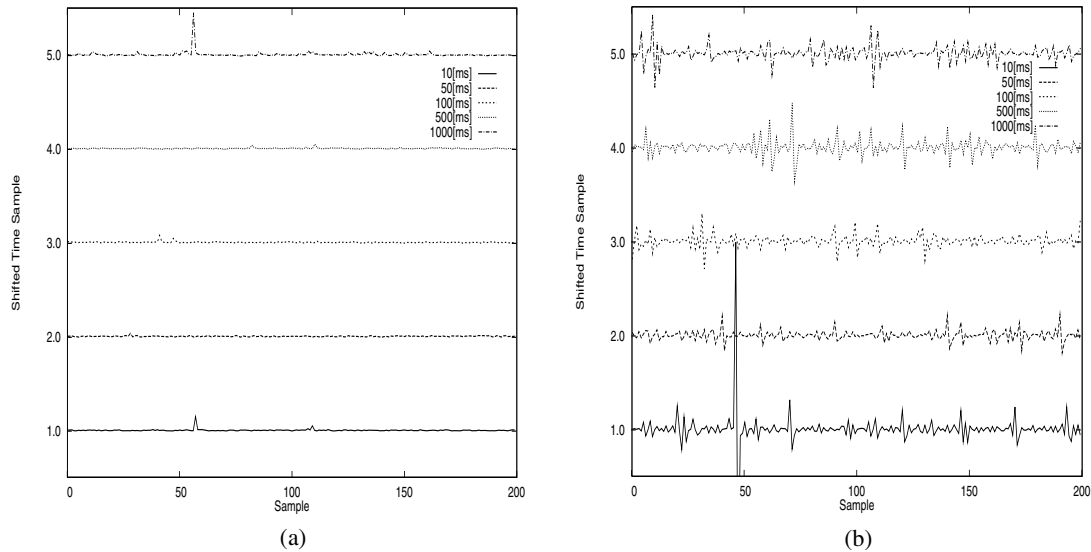


Figure 4. (a) ARIADNE versus (b) Unix timers.

can be visualized and compared on the same scale. Each such graph can be represented as a vertically shifted function $y'(x)$, where

$$y'(x) = y(x) - \bar{y} + c$$

In this equation, $y(x)$ represents the original set of measurements, \bar{y} is the average of these measurements, and c is an integer constant in the interval $[1, 5]$. The timing offset of roughly 10 ms that was observed in the set of Unix timer results shown in Figure 3(b) were also observed in the experimental measurements that resulted in Figure 4(b). This timing offset was of roughly the same magnitude (i.e. 10 ms) for all the timer settings that we experimented with. In comparison, ARIADNE's timers were very precise, consistently exhibiting an offset in the order of roughly 200 μ s from the actual time requested in the worst case; the average timing offset is roughly 30 μ s.

Experiment 2: load effects

To study how timers may be affected by processor loads we repeated the experiments described above under various load conditions. We generated a controllable external load by creating processes that run continuously in an infinite loop. The processor load units shown on the x -axes of Figures 5, 6, and 7 represent the number of load-generating processes in the system when the corresponding timer measurement is made. For each processor load, we collected 200 samples of timer measurements (i.e. actual elapsed time to detection of timer expiry), with the sample averages displayed on the y -axes of the graphs. Each average is displayed along with a 90% confidence-interval based on a Student- t distribution. In the case of Figures 5(a) and 6(a), the load-generating processes ran at the

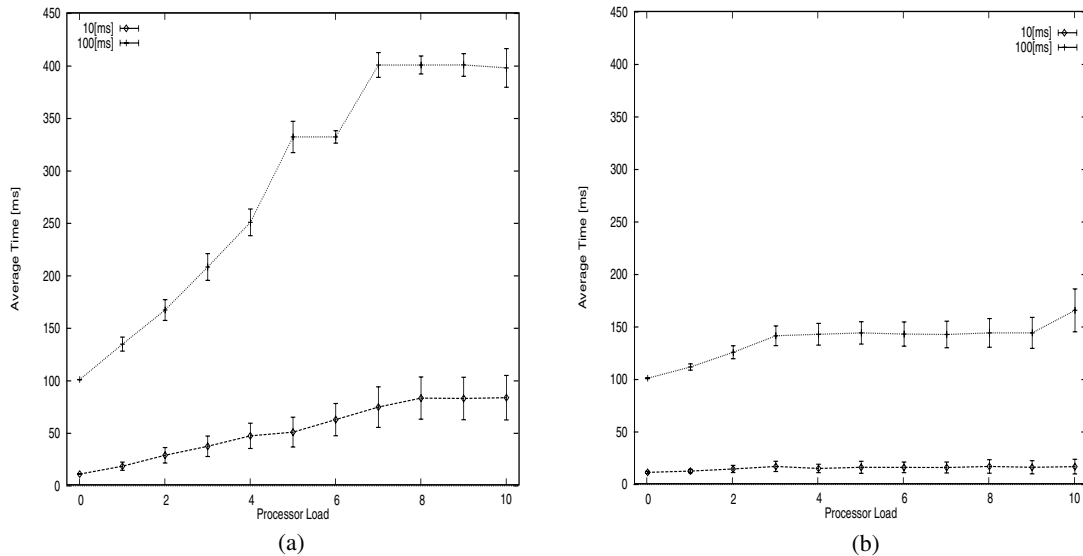


Figure 5. ARIADNE timers results for precision versus load: (a) same priority load; (b) low priority load.

same priority as the process that ran the timing code. The results shown in these graphs suggest that although ARIADNE's timers are precise, timer quality is negatively effected by increasing process loads and competing processes that run at the same priority as the measuring process (see Figure 5(a)). In contrast, the Unix timer (see Figure 6(a)) appears to remain stable even in the presence of increasing process loads. Observe, however, that the Unix timer measurements still exhibit a significant timing offset (i.e. 10 ms) with respect to the actual values at which the timer was set (10 ms and 100 ms, respectively).

We explain the significant differences between ARIADNE's timing behavior and Unix's timing behavior as follows. With ARIADNE, the load-generating processes and the measuring process ran at the same priority, and both types of processes were CPU bound. Unix's scheduler gives these processes equal treatment, and allocates them to the CPU in a round-robin fashion. The higher the number of load-generating processes, the longer the period during which the measuring process gets displaced from the CPU. As a result, the average measured elapsed-time increases linearly with the number of load-generating processes, and the measured timing offset increases linearly with the length of a round-robin scheduling cycle.

In the case of the Unix timer measurements, however, the measuring process is not CPU bound (see Figure 6(a)). This process goes into a blocked-wait mode when it invokes the `pause()` system call, blocking in the kernel until its signal arrives. As a result, the Unix scheduler actually gives it a higher priority than the load-generating CPU bound processes. Due to its higher priority, the measuring process is scheduled with preference over the load-generating processes, and actually preempts the

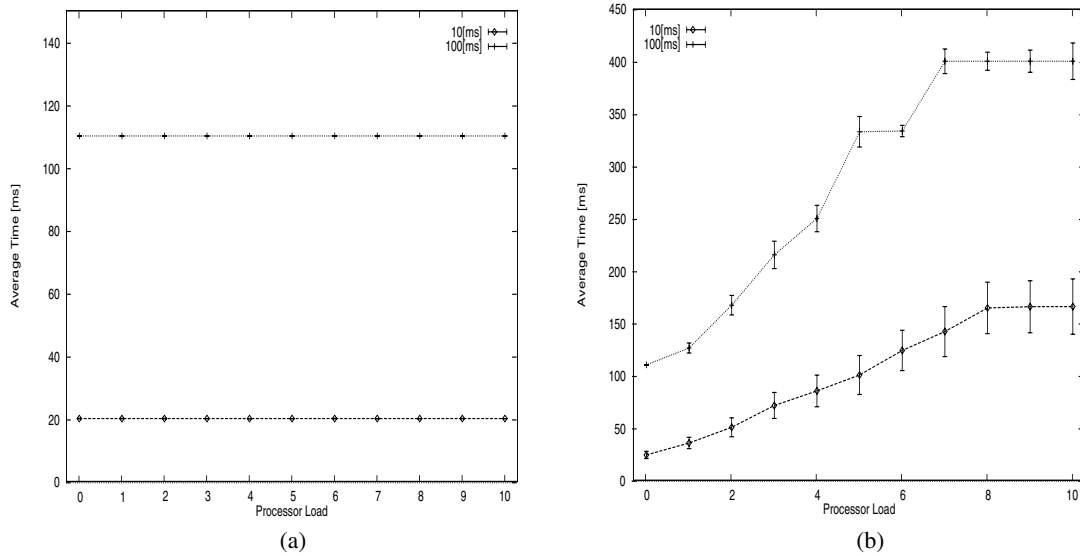


Figure 6. Unix timers results for precision versus load: (a) block-wait version; (b) busy-wait version.

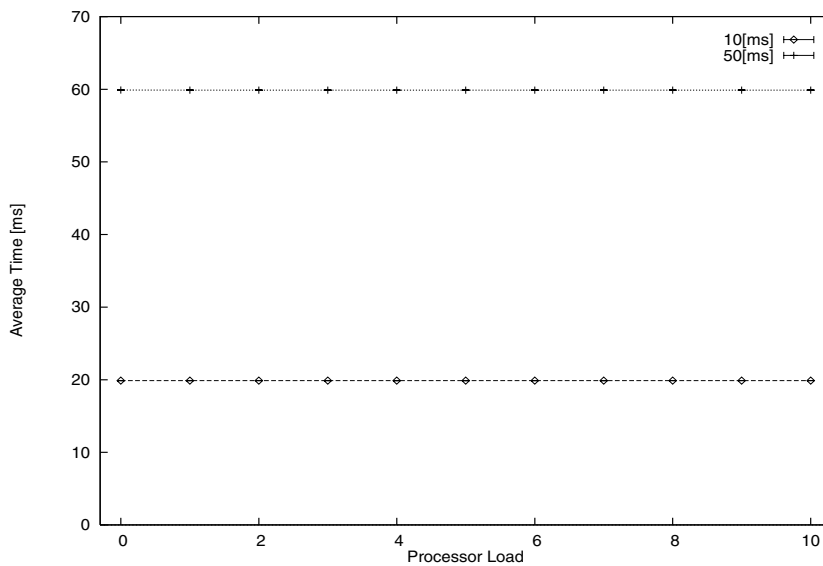


Figure 7. Unix timers results for precision versus load (real-time scheduling priority).

other processes whenever its timer expires. However, even so, it is still not able to eliminate the relatively large timing offsets.

If the priority of the measuring process in ARIADNE's timer test is increased with respect to the priority of the load-generating processes, we will achieve the same effect as witnessed in the Unix timer case. In other words, ARIADNE's timer will experience an increasing amount of independence from the load-generators as the measuring-process's priority increases. This effect can be seen in Figure 5(b), with results obtained using load-generators running at a *nice* level that is 19 units higher than the *nice* level of the process doing the ARIADNE timer measurements. This effectively increases the priority of the measuring process with respect to the other processes, and yields more accurate actual elapsed-times. Although ARIADNE's processes can be made to block whenever they have no useful work to do, blocking can result in poor performance when load is low.

In Figure 6(b) we see the results of a similar experiment in which the measuring process for the Unix timer measurement is forced to be CPU bound. Instead of using the `pause()` system call to wait timer expiry, the process is made to spin in a busy-wait cycle until it receives the timing signal. In this case, we witness an effect with Unix's timer that is similar to the effects shown in Figure 5(a), with ARIADNE's timer. There is, however, an important difference: Unix's timer exhibits a timing offset of roughly 10 ms in all our measurements involving low-load conditions. The quality of the timer deteriorates for precisely the same reasons given in the discussion of the results shown in Figure 5(a). Since the load-generators and the measuring process for the Unix timer measurements are all CPU bound and run at the same priority, they end up being scheduled in round-robin fashion. Elapsed times that are greater than the time-quantum allocated to the measuring process are likely to incur a timing offset that corresponds to the sum of the quanta of all the load-generators in the system.

If the measuring process for the Unix timer measurement is scheduled under the real-time scheduling class[†], timing measurements cease being susceptible to variations in load that is generated by processes scheduled under other scheduling classes. This effect is shown in Figure 7. Note, however, that the timing offset of roughly 10 ms, which depends on the clock-tick rate in the system configuration, still persists.

CONCLUSIONS

With a goal of delivering an efficient mechanism for incorporating asynchronous signals into a threads system, we have managed to arrive at an efficient implementation of precise timers in the ARIADNE user-level threads library. In situations of low processor loads, we have found the proposed timing scheme to be very precise. However under high processor loads, however, the ARIADNE timing scheme degrades if processes with the same priority as the ARIADNE timing process compete for CPU attention; this occurs because an ARIADNE process, supporting user-level threads, runs at the mercy of the operating system scheduler. By increasing an ARIADNE process's priority we may effect an improvement in timer accuracy, but this is not a viable solution in most operating systems unless ARIADNE processes run with super-user privileges. Given that our application domain includes

[†]The use of this feature requires special system privileges.

threads-based user-level protocols for heterogeneous distributed computing, the acquisition of super-user privileges on widely distributed systems, which are typically under the management of different administration domains, is unrealistic. Hence, it is not reasonable to expect that a solution that is based on increasing a process's priority will work for distributed applications such as CLAM, which exploits hosts located in different administration domains.

In addition to timer design and experimentation, we have described an extended and flexible user interface to ARIADNE's timer subsystem and signal-activated threads. We based this interface on the requirements specified by the CLAM multiprotocol system. We have used ARIADNE's timer interface extensively in the support of failure recovery in CLAM. Further, signal-activated threads play a key role in implementing asynchronous network I/O for CLAM's embedded protocol modules. This interface has also been used in the CLAM process management module for the purpose of monitoring process-state when multiple processes are created on a given host. We have yet to explore how continuations can improve protocol performance in CLAM, and how they may be used to shorten critical sections in ARIADNE. We expect that performance improvements in this area will further increase the efficiency of CLAM's user-space protocols, making them even more competitive with respect to in-kernel protocols.

ACKNOWLEDGEMENT

JCG's research was done while a PhD student at Purdue University, West Lafayette, IN, U.S.A. Research supported in part by DoD DAAG55-98-1-0246 and PRF-6903235.

REFERENCES

1. Gomez J, Rego V, Sunderam V. Efficient multithreaded user-space transport for network computing: Design and test of the TRAP protocol. *Journal of Parallel and Distributed Computing* 1997; **40**(1):103–117.
2. Gomez J, Mascarenhas E, Rego V. The CLAM approach to multithreaded communication on shared-memory multiprocessors: Design and experiments. *IEEE Transactions on Parallel and Distributed Systems* 1998; **9**(1):1–14.
3. Mascarenhas E, Rego V. Ariadne: Architecture of a portable threads system supporting thread migration. *Software—Practice and Experience* 1996; **26**(3):327–357.
4. Gomez J, Rego V, Sunderam V. CLAM: Connectionless, lightweight, and multiway communication support for distributed computing. *Communication and Architectural Support for Network-based Parallel Computing (Lecture Notes in Computer Science, vol. 1119)*. Springer: Berlin, 1997; 227–240.
5. Yates D, Nahum E, Kurose J, Towsley D. Networking support for large scale multiprocessor servers. *Proceedings of ACM SIGMETRICS*, 1996. ACM Press: New York, 1996.
6. Nahum E, Yates D, Kurose J, Towsley D. Performance issues in parallelized network protocols. *Proceedings of the 1st USENIX Symposium on OSDI*. ACM/SIGOPS, 1994; 125–137.
7. Sanghi D, Subramaniam MCV, Shankar U, Gudmundsson O, Jalote P. A TCP instrumentation and its use in evaluating round-trip time estimators. *Internetworking: Research and Experience* 1990; **1**(2):77–99.
8. Varghese G, Lauck T. Hashed and hierarchical timing wheels: Data structures for efficient implementation of a timer facility. *ACM SIGOPS Operating Systems Review* 1987; **21**(5):25–38.
9. Karn P, Partridge C. Improving round-trip time estimates in reliable transport protocols. *Computer Communicators Review* 1987; **17**(5):2–7.
10. Jacobson V. Congestion avoidance and control. *Proceedings of the ACM SIGCOMM '88 Symposium*, August 1988; 314–329.
11. Jacobson V, Floyd S, Estrin D, Sharma P. Scalable timers for soft state protocols. *Proceedings of IEEE INFOCOM'97*. IEEE Computer Society Press: Los Alamitos, CA, 1997.
12. Zhang L. Why TCP timers don't work well. *Proceedings of the ACM SIGCOMM '86 Symposium on Communications Architectures and Protocols*, Stowe, VT, 5–7 August 1986. *Computer Communication Review* 1986; **16**(3):397–405.

13. Holbrook H, Singhal S, Cheriton D. Log-based receiver-reliable multicast for distributed interactive simulation. *ACM SIGCOMM* 1995; **25**(4):328–341.
14. Jain R. Congestion control in computer networks: Issues and trends. *IEEE Network Magazine* 1990; (May):24–30.
15. Buskens R, Siddiqui M, Paul S. Reliable multicast of continuous data streams. *Bell Labs Technical Journal* 1997; **2**(2):151–174.
16. Paul S. RMTP: A reliable multicast transport protocol. *Proceeding of IEEE INFOCOM'96*. IEEE Computer Society Press: Los Alamitos, CA, 1996.
17. Kleiman S, Shah D, Smaalders B. *Programming with Threads*. SunSoft Press/Prentice-Hall: Englewood Cliffs, NJ, 1996.
18. Anderson T, Bershad B, Lazowska E, Levy H. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems* 1992; **10**(1):53–79.
19. Mascarenhas E, Rego V. Migrant threads on process farms: Parallel programming with Ariadne. *Concurrency—Practice and Experience* 1998; **10**(9):673–698.
20. Wallach DA, Hsieh WC, Johnson KL, Kaashoek MF, Weihl WE. Optimistic Active Messages: A mechanism for scheduling communication with computation. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOP)*, July 1995; 217–225.
21. Draves R, Bershad B, Rashid R, Dean R. Using continuations to implement thread management and communication in operating systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles, Operating Systems Review*, Pacific Grove, CA, October 1991; 122–136.
22. Dean R. Using continuations to build a user-level threads library. *Proceedings of the 3rd USENIX Symposium on MACH OS*, April 1993. USENIX Association: Berkeley, CA, 1993; 137–152.