

# Experimentation with Configurable, Lightweight Threads on a KSR Multiprocessor

*Kaushik Ghosh*

*Bodhisattwa Mukherjee*

*Karsten Schwan*

June 23, 1993

College of Computing  
Georgia Institute of Technology  
Atlanta, Georgia 30332  
e-mail: {bodhi, kaushik, schwan}@cc.gatech.edu

**GIT-CC-93/37**

*6 June 1993*

## **Abstract**

The implementation of operating system functions can significantly affect the performance of parallel programs. Our research concerns the customization of operating system functionality for different target hardware to improve the performance of application programs. In this paper, we describe our experience with a reconfigurable, multiprocessor Mach cthreads package on a 32-node KSR-1 supercomputer. Sample static

and dynamic configurations address the exchange and on-line adaptation of threads schedulers, and the on-line adaptation of threads synchronization constructs. Experimental results are demonstrated with two different parallel application programs, (1) a parallel branch-and-bound application and (2) the runtime kernel of a Time Warp discrete event simulator. The lightweight threads package has been ported to several target architectures, including Sparcstations, a 32-node GP1000 BBN Butterfly, SGI multiprocessors, and the 32-node Kendall Square Supercomputer.

College of Computing  
Georgia Institute of Technology  
Atlanta, Georgia 30332-0280

# 1 Introduction

Lightweight threads packages have been built for a variety of parallel and sequential machines. Typical arguments for their use include complete user-level control over threads scheduling[BLL88] and possibilities regarding the customization of threads synchronization[ALL89] or communication constructs[BALL90]. Our research requires user-level lightweight threads for two reasons: (1) to permit the customization of threads package functions to specific target applications, and (2) to perform such customization on-line in conjunction with the application's execution.

This paper describes the basic functionality and performance of a configurable lightweight threads package on a KSR shared memory supercomputer. Lightweight threads' performance is compared to the performance of the kernel-level threads offered by the machine's manufacturer (pthreads). Next, the on-line configuration of the threads package's mutex locks<sup>1</sup> is shown to significantly improve the performance of representative parallel application programs on the KSR machine. Last, we describe a design of a configurable threads scheduler able to improve program performance by changing selected scheduler attributes during program execution.

## 2 Previous Results

We have already investigated threads-level configuration for a specific construct – reconfigurable locks<sup>2</sup> – used for the synchronization of multiple processes on the BBN Butterfly NUMA multiprocessor[MS93a, MS93b]. Using artificial workloads[MS93a] and a representative multiprocessor application – a parallel branch-and-bound program – on a 32 node BBN Butterfly multiprocessor[Muk91], we have demonstrated that:

- Off-line configuration is important for use of alternative thread schedulers, such as for real-time vs. non-real-time applications[SZG91, GFS93a]. Similarly, multiple thread schedulers may be employed concurrently for performing the scheduling of application threads vs. threads performing communication processing[CMS93, LKAS93].
- Performance improvements can be attained by adapting the scheduling components of a lock construct to the application's dynamically changing locking pattern (e.g., access frequencies, lengths, and contention on critical sections), ranging from improvements of 3-5% for programs not exhibiting significant lock contention to 17% for applications making frequent use of single, shared queues.
- The functionality of the resulting threads package-level “adaptive objects” is comprised of (1) methods for monitoring object state, (2) methods for changing selected attributes of the object's internal representation, including its scheduling component, and (3) methods for the mutually exclusive acquisition of such attributes by external agents desiring to perform a change.

These results are evidence of the utility of threads-level configuration, on-line or off-line. In the remainder of this paper, we first mention a few salient aspects about the operating system on

---

<sup>1</sup>The terms ‘mutex lock’ and ‘blocking lock’ have been used interchangeably in this paper.

<sup>2</sup>The terms ‘reconfigurable lock’ and ‘adaptive lock’ have been used interchangeably in this paper.

the KSR machine (OSF's Mach), and review the basic structure and performance of lightweight threads on the machine. Next, selected aspects of the internal structure of the threads package permitting the configuration of selected constructs are reviewed, using the example of scheduler configuration for real-time applications and for communication processing. Results concerning runtime configuration of threads library constructs focus on lock reconfiguration for the branch-and-bound application and the Time Warp kernel. Last, a design and implementation of on-line scheduler configuration for optimization of optimistic discrete-event-simulation programs is described.

### **3 The KSR: Architecture, and Basic Performance Measurements of Pthreads and Cthreads**

To put later performance measurements in perspective, in this section we review the architecture of the KSR, report the performance of a few commonly used OS and pthread library calls, and then the timing measurements of some of the most commonly used cthreads calls.

#### **3.1 Architecture**

The KSR supercomputer is a shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes or 34 rings (the largest machine delivered to date consists of 256 processors). Each node consists of a 64-bit processor, 32 MBytes of main memory used as a local cache, a higher performance 0.5 MBytes sub-cache, and a ring interface. CPU clock speed is 20 MHz, with peak performance of 40 MFlops per node, an access time to the subcache of 2 processor cycles (with a 64-byte cache line), an access time of 18 processor cycles to local memory, and an access time of 126 cycles to remote memory using a 128-byte cache line. Therefore, severe penalties exist concerning accesses to sub-cache, cache, and remote memory. Such penalties increase when additional rings exist in the memory access hierarchy.

Programmers do not perceive the memory hierarchy existing in the machine (other than by potentially observing performance penalties), since access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheap. Furthermore, at the lowest level, the parallel programming model offered by the KSR's OSF Unix operating system is one of kernel-level threads, called pthreads, offering constructs for thread fork, thread synchronization, shared memory between threads, etc. pthread functions are accessed via kernel calls, and much of the operating system kernel's code, state, and buffer space is replicated across the different processor caches, thereby resulting in local access to operating system functionality.

## 3.2 Measurements of the Basic Functions

The measurements reported below were taken on a 32-node machine<sup>3</sup>. The numbers quoted here are the averages over 10,000 to 100,000 repetitions for each primitive.

### 3.2.1 KSR OS and Pthreads

As in most Unix-like operating systems, the KSR OS provides operations to save and restore the machine context. A context save operation requires 5.34 microseconds when the signal mask need not be saved<sup>4</sup>, and 269.48 microseconds when it is saved<sup>5</sup>. Restoring context takes an almost identical amount of time: 6.17 microseconds without signal mask<sup>6</sup>, and 215.24 microseconds when the mask is saved<sup>7</sup>.

A basic atomic instruction (`gspnwt`:get sub-page without waiting) requires 1.32 microseconds and a null function call may be performed in 1.78 microseconds.

The graphs below show the performance of dynamic memory allocation on the KSR. Figure 1 shows the timing measurements when memory is allocated from a prepagged pool. This is useful while allocating stack for threads, e.g., because the performance hit due to the first page fault need not be incurred. Figure 2 shows the corresponding numbers for non-prepagged memory (which are much greater than prepagged memory). From the measurements, there appears to be a minimum chunk size ( $2^7$  bytes) for memory allocation on the KSR.

Creating a pthread requires 3286.92 microseconds, and a create-and-bind of a pthread to a processor requires 9984 microseconds. Yielding the processor requires 100.12 microseconds when there is a single pthread in the run-queue, and 179.56 microseconds when there are two pthreads (ping-pong). A lock-and-unlock of a pthread mutex requires 10.93 microseconds.

### 3.2.2 Lightweight Threads on KSR

The lightweight threads package constructed by our group uses a single pthread as a virtual processor for the execution of multiple user-level threads (we often call these user level threads ‘cthreads’). Essentially, each pthread simply runs any thread it finds in its local ready queue of user-level cthreads. The basic functions offered by the user-level threads functions are those of Mach CThreads[CD88], including thread management, synchronization, and memory management. Thread initialization differs from Mach cthreads in that users can control the total amounts of shared memory allocated for threads and thread stacks, the number of virtual processors (pthreads) used for thread execution, etc. Memory management routines are mapped directly to the KSR operating system’s routines, in contrast to versions of cthreads constructed for other machines (e.g., the BBN Butterfly or Sparcstations). We are currently experimenting with a buddy allocator, which will allow applications to vary the minimum chunk of memory allocation for performance improvement.

The cthread implementation has not been optimized for the KSR machine. It is derived from a similar implementation being used in our research on SUN Sparcstations.

---

<sup>3</sup>The measurements were taken when the load on the 32-node KSR was between 15 and 20.

<sup>4</sup>This is nothing but the `_set jmp` operation. 3

<sup>5</sup>This is the `set jmp` operation.

<sup>6</sup>This is the `_long jmp` operation.

<sup>7</sup>This is the `long jmp` operation.

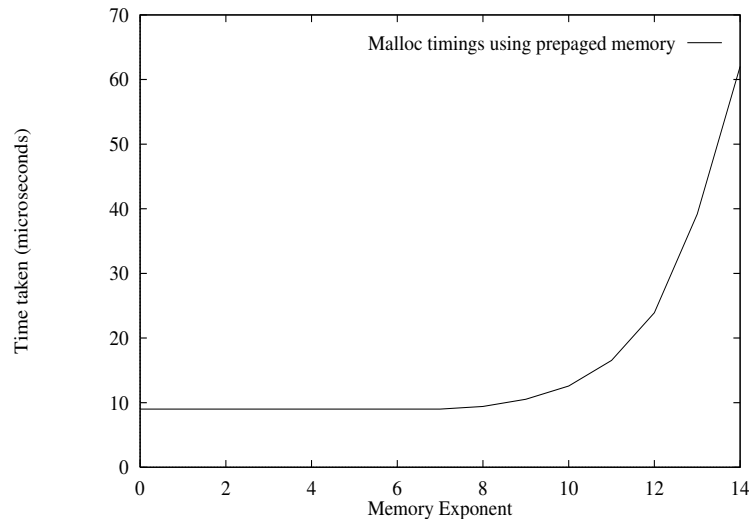


Figure 1: Time for malloc using pre-paged memory (microseconds)

Forking a cthread on the local processor takes 71.5 microseconds, and forking on a remote processor takes 120.58 microseconds. A fork-and-detach of a cthread requires 378.37 microseconds. It takes 38.69 microseconds to yield the processor when there is one cthread in the run queue, and 78.09 microseconds when there are two (ping-pong). A spin-lock-and-unlock requires 4.71 microseconds in our cthreads library, and a mutex-lock-and-unlock takes 6.19 microseconds.

The performance measurements that can be compared for pthreads and cthreads are listed in table 1.

Operation	pthread	cthread
fork	3286.92	71.5
yield	100.12	38.69
ping-pong	179.56	78.09
mutex lock+unlock	10.93	6.19

Table 1: Summary of the performance of pthreads and cthreads on the KSR (microseconds)

There is scope for improvement in some of the functions reported above. Context-switch could, for example, be configured to the individual needs of the application: all applications might not use the full set of registers on the KSR (32 CEU, 32 IPU, 64 FPU registers, apart from the XIU registers). We are going to undertake such investigations in our future work.

Although the measurements shown above do not present all performance advantages afforded by use of lightweight vs. kernel-level threads on the KSR machine (other performance advantages

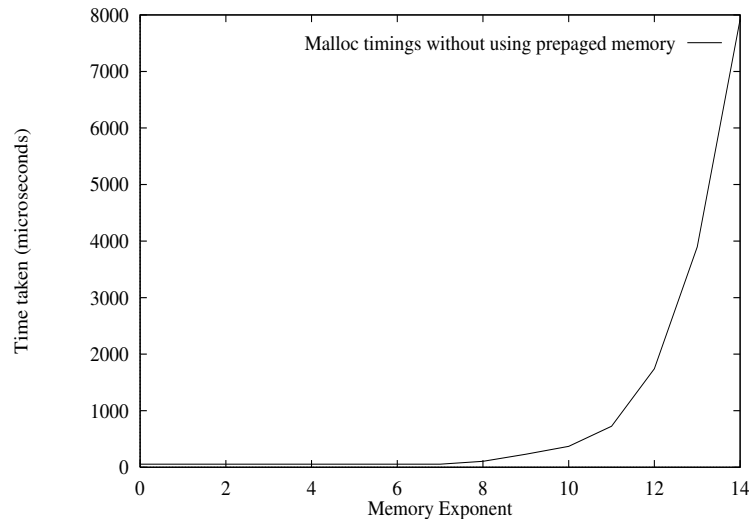


Figure 2: Time for malloc using non-pre-paged memory (microseconds)

derive from savings in lock or condition allocation times, etc.), it is apparent that programs wishing to map multiple user threads to a single processor can derive significant benefits from using cthreads vs. pthreads.

## 4 Applications Running with Cthreads on KSR

In this section, we present the salient points about two applications running on cthreads on the KSR: a Travelling Sales Person application, and the kernel of a Time Warp simulator, thus setting up a ‘control’ scenario for performance comparison with an implementation using reconfigurable locks, described later in this paper.

### 4.1 Time Warp with Pthreads vs. Cthreads on KSR

Time Warp is a discrete-event simulation technique that uses optimistic (rather than conservative) parallelization techniques [Jef85]. Salient features of the Time Warp testbed that we use for our research have been treated extensively elsewhere [Fuj89]. Here, we review some of the parts of the kernel where locking is performed.

- When a logical process (LP) sends a message to another LP (or to itself), the list of free events, and the list of incoming events of the recipient’s processor are locked. (These lists are on a per-processor basis.)

- When an event finishes execution, the corresponding processor examines its list of incoming events; this entails locking that list.
- Before putting an event for cancellation in the list of the appropriate processor, the corresponding cancellation list has to be locked.
- Before cancelling events, the cancellation list on that processor has to be locked to get a handle on the events to be cancelled.
- When an unprocessed event is cancelled, it is returned to the free list of events; this involves locking the list of free events.
- When the simulator runs out of free memory for free events, it undertakes garbage collection. Here, the processors have to examine the event lists, the list of events candidate for cancellation, the list of messages sent but not yet received, and the list of events candidate for rollback; all these data structures have to be locked for this purpose (however, garbage collection is performed relatively infrequently). Further, the list of free events has to be locked when garbage-collected events are returned to the free list.

In this paper, we report the performance of the kernel in response to a Parallel Hold workload model, the generality of which has been argued elsewhere [Fuj90]. We use a single cthread on each processor; all the logical processes (LPs) mapped on that processor are multiplexed by this cthread.

The following graphs show the performance of the kernel using the pthreads library, and our cthreads library. We used blocking (also called mutex) locks in the cthreads implementation. When a cthread tries to get a mutex lock and fails, it yields the processor before trying again. Since there was only one cthread per processor, that thread will start to run again, but there will be a considerable delay before it tries to get the lock again<sup>8</sup>.

It should be noted that we started the ‘wall clock’ *after* forking the pthreads/ctthreads on the different processors. Initializing the cthreads library takes some time, but forking a cthread is so much cheaper than forking a pthread that the initialization cost is quickly amortized if several threads are forked on each processor. While the current implementation of the Time Warp kernel uses one thread per processor, it should be noted that there are several applications that fork multiple threads on each processor, and even dynamically fork threads. Such applications would greatly benefit from a cthreads-like library (since forking is much cheaper there than pthreads). The current implementation of the Time Warp kernel performed better with pthreads than with cthreads chiefly because locking is cheaper for cthreads.

## 4.2 Travelling Sales Person with Cthreads on KSR

We have implemented the TSP algorithm as a collection of asynchronous cooperating threads[CMS93]. The threads cooperate through two shared abstractions: (1) a shared work queue of subproblems which stores the search-space tree, and (2) a shared value representing the current best tour found so far. To start a computation, the main thread of the program first enqueues the

---

<sup>8</sup>Recall that the time for a mutex lock followed by an yield is about 40 microseconds (38.69 microseconds for yield, and about 6.19/2 microseconds for mutex lock) for cthreads; compare this with 4.71/2 microseconds required for a spin lock: the interval between successive attempts to lock would be lower had spin locks been used. Spin locks would increase contention on the slotted ring (connecting the processors) in case of enough contention on the locks.

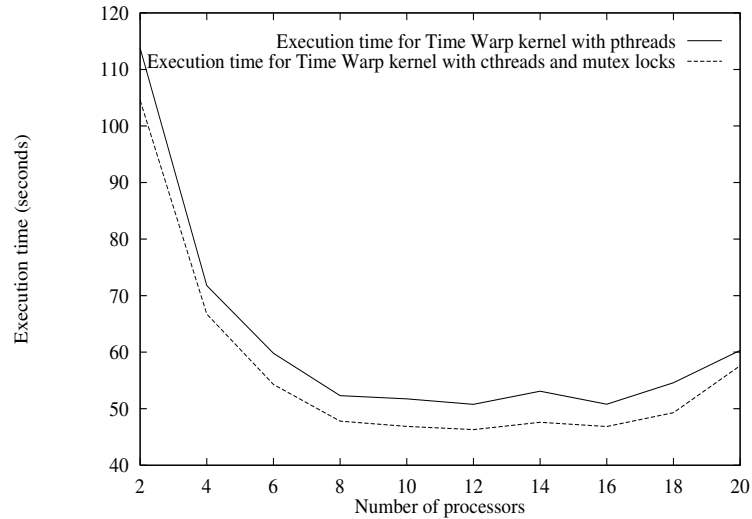


Figure 3: Execution time for Time Warp with pthreads, and with cthreads using mutex locks (seconds)

node of the initial problem (the root node) in the work queue, it then forks a number of searcher threads and finally waits until all of them terminate. A searcher terminates when at least one tour has been found and there is no unexplored node in the work queue.

The implementation uses four synchronization locks – `qlock` for mutual exclusion of the shared queue, `glob-act-lock` for mutual exclusion of the variable containing the number of active slaves, `glob-low-lock` for mutual exclusion of the lowest tour value (for the distributed representation, each processor keeps its own local copy of the value), and `globlock` which is a multi-purpose lock to keep the global data structure consistent.

Figure 4 shows the performance of TSP for varying number of processors. While we did not carry out a pthreads implementation, the performance of the Time Warp kernel (using cthreads) should convince the reader that other applications (including TSP) would perform well when compared with pthreads on the KSR.

However, the object of our research is not simply to gain such performance benefits. We next motivate the need for off-line threads package configuration, using the example of scheduler configuration.

## 5 Configuration of Lightweight Threads – Scheduler Configuration

Any configurable thread scheduler may be described as an object offering certain operations and attributes that vary due to differences in internal scheduler representation. Such attributes must

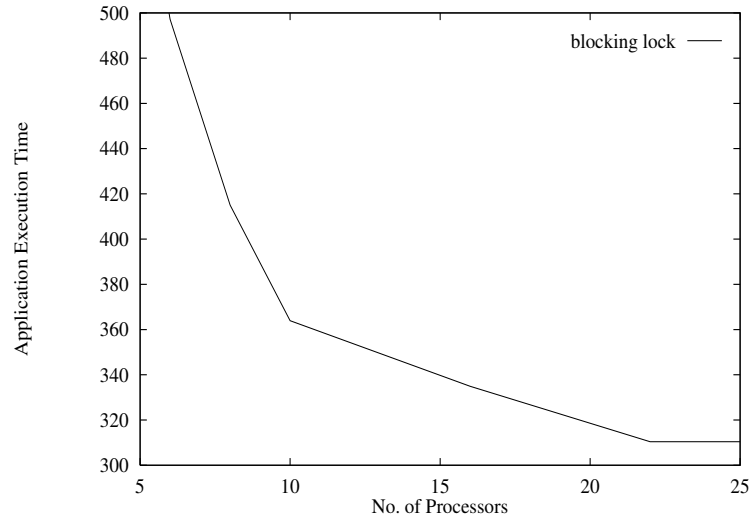


Figure 4: Execution time for TSP using cthreads with blocking locks (seconds)

capture the following implementation differences:

- *Granularity*: Each scheduler schedules a unit of activity at a time. The granularity of scheduling defines the size of such a unit, which may consist of a single or a group of threads.
- *Storage/retrieval policies*: The scheduler’s repository for schedulable (ready to run) threads will differ among scheduling policies, ranging from simple FCFS queues to priority- or even deadline-ordered queues[SZG91].
- *Locality*: Scheduler performance is also affected by the distribution of the above-mentioned thread repository. Especially on the KSR machine, global queue organizations are not suitable for scalable performance in thread scheduling and thread management, so that queues must be internally fragmented across different processors[CMS93]. Additional scheduler attributes must capture whether threads may be migrated across processors, which affects load balancing and cache performance.
- *Preemption*: Scheduling policies may be classified as enforcing (1) involuntary preemption, which means that a scheduler can preempt a thread at any time, (2) voluntary preemption, which means that threads must release processors voluntarily, and (3) non-preemptive, which implies that threads are never preempted. Such differences result in major changes in the implementation of thread context switching (saving or not saving signal masks, for instance); they occur for real-time vs. non-real-time schedulers.
- *Application hints*: Schedulers must be able to offer attributes that may be specified by application programs, such as handoff hints[Bla90] or real-time parameters like deadlines.

We next describe the class definition of a scheduler offering attributes for its static configuration as a preemptive (using signalling) vs. non-preemptive scheduler. These class definitions are used in the instance of lightweight threads simultaneously using two different schedulers, one running communication threads, the other running application threads. These schedulers are used in a distributed implementation of a “work queue” abstraction in a parallel branch-and-bound code. Significant performance improvements are demonstrated when using a non-preemptive FIFO scheduler for application threads and simultaneously using a preemptive priority scheduler for asynchronously executable threads implementing the communications among the different fragments of the work queue abstraction.

CLASS *Scheduler* is

```

STATE internal_state is
    LOCK sched_lock;
    int node;
    int active_threads;
    QUEUE_T free_list;
    QUEUE_T *lqueue;
    THREAD_T scheduler_thread;
END

ATTRIBUTESET static_scheduler_attributes is
    int no_of_readyqs;
    int no_of_threads_per_proc
    HINT_T hint;
    OPERATION global_init (..);
    OPERATION vproc_init (..);
    OPERATION thread_init (..);
    OPERATION get_thread (..);
    OPERATION put_thread (..);
    OPERATION empty_readyq (..);
    OPERATION proc_idle (..);
    OPERATION schedule (..);
    OPERATION alert_hdlr1 (..);
    .
    .
    OPERATION alert_hdlrn (..);

BEGIN
END

```

The above class defines an application-specific scheduler. The internal state of the scheduler provides data structures to define/implement multiple scheduling policies. The `attributeset` of

a scheduler consists of various levels of initialization operations (`global_init` initializes at the processor level, `vproc_init` initializes at the virtual processor level, and `thread_init` initializes at the thread level), different policy operations (`get_thread` to get the next thread to run, `put_thread` to put a thread back to a bank of ready queues, `empty_readyq` to determine if a ready queue is empty, `proc_idle` to specify work for an idle processor), and a set of operations to handle various exceptions/interrupts (primarily used to implement preemptive scheduling based on inter-processor interrupts, timer interrupts etc.).

Such a scheduler framework can be easily used to define any application specific scheduler (possibly different schedulers on different processors running an application at the same time<sup>9</sup>). An application defines the scheduler(s) at pre-execution time and installs them to the specific processors at the application-initialization time using a call to `install_scheduler(processor number, scheduler)`.

A real-time application, e.g., may not want to use the generic function in the `cthrads` library, whereby a FCFS policy is used to run threads until they explicitly yield. Real-time systems often use variations of priority-driven scheduling (for example Earliest Deadline First or multiprocessor variants thereof [CC89, SZG91, GFS93a]) in the absence of sporadic task arrivals, and perform schedulability analyses whenever a sporadic arrives (and interrupts the processor). From the aforementioned text, it should be clear that such application-specific schedulers are easy to install in the `cthrads` library. This would constitute static scheduler configuration of the library. Dynamic scheduler reconfiguration is discussed in a later section.

## 6 Runtime Configuration of Lightweight Threads – Synchronization

Any dynamically configurable object defines the abstraction-specific attributes mentioned above. At runtime, such attributes are altered explicitly or implicitly by a user-provided “adaptation policy”. We assume that changes in object attributes may be performed both synchronously or asynchronously with method invocations. This requires the introduction of two additional time-dependent properties of object attributes: (1) attribute mutability and (2) attribute ownership. An attribute is *mutable* whenever its current value may be changed. For example, a lock object’s attribute specifying its waiting policy (not its scheduling policy) is permanently mutable because it may be changed at any time. However, its scheduling policy is likely to be immutable whenever threads are waiting on the lock, due to the inordinate potential expenses involved with the reorganization of internal lock data structures like thread waiting.

Since runtime configuration requires state monitoring, the `cthrads` package provides a set of functions and facilities for on-line monitoring not described in detail here (see [OSS90] for a discussion of an earlier implementation of such primitives performed by our group). Such information (about threads package state) may be used at the application level by “adaptation policies” applied to multiple threads package components (configurable objects), or it may be used to build single adaptive components. For example, the adaptive lock object used in the experimentation below monitors lock contention (in terms of number of waiting threads) periodically at a user-defined

---

<sup>9</sup>We call such scheduling schemes “Heterogeneous Scheduling”.

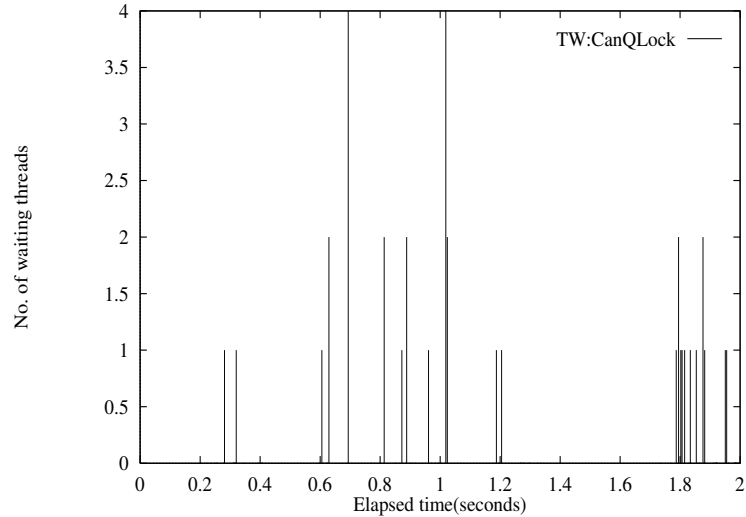


Figure 5: Contention patterns for lock on list of events to be cancelled in Time Warp; 10-processor run

rate, and adjusts several of its implementation-specific attributes (i.e., its `spin-time`, `sleep-time`, `release-time` etc.) according to an adaptation policy. The adaptation policy used in the experiments below is to dynamically change the locks to spin locks when lock contention is low, and to change them to blocking locks when it is high. The actual attributes varied are the `spin-time` and `sleep-time`. This implies that the lock dynamically changes its waiting policy from pure spin-waiting, to back-off waiting, to blocking depending on the application’s current locking behavior.

We perform runtime lock configuration on two ‘applications’: a TSP program, and the Time Warp kernel referred to earlier. The object of these experiments is to demonstrate that the performance of a complex parallel program can be improved without changes to the program, but simply by adapting the locking primitives employed by this program.

We monitor each application to ascertain whether there was any contention on the locks, and if so, how much of contention there was. As stated earlier, we use the number of threads waiting on a lock as a metric in this regard. Associated with each reconfigurable lock is a counter. When a cthread fails to acquire a lock, it increments that counter. The value of the counter and the corresponding ‘real-time’ together provide a profile of the contention pattern on the lock. The customized lock monitor for the adaptive locks, used in the experiments, senses the number of waiting threads (sampled once during every other unlock operation) to be used by the adaptation module.

Low contention suggests that blocking locks can be changed to spin locks, since that lock is probably immediately available. This is done by changing the sleep-time of the lock to a very small value, and the spin-time to a high value. High contention would suggest performance improvement by changing a spin lock to a blocking lock: its sleep-time increased and its spin-time reduced. Further, as the contention on a lock changes over the lifetime of the application, these two parameters

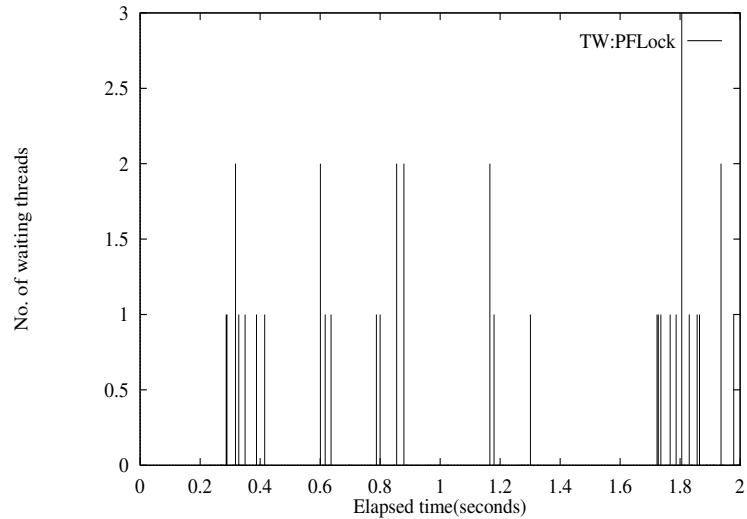


Figure 6: Contention patterns for lock on free list of events for Time Warp; 10-processor run

can be continuously changed to provide a spectrum of locks from ‘pure-spin’ to ‘pure-block’.

We monitor three locks in both applications. In the Time Warp kernel, they are the locks on the cancellation list, the event list, and the lock on the free list of events. The contention patterns are reproduced for the Time Warp and the TSP applications<sup>10</sup>

The Time Warp kernel shows relatively little contention on its locks<sup>11</sup> (figures 5, 6, 7). Therefore, we use spin locks in its ‘reconfigured’ implementation. The TSP application, shows a high degree of contention (figures 8, 9, 10). After studying the locking patterns for all the locks, we found that **glob-low-lock** suffers from almost no contention, and **globlock** and **glob-act-lock** suffer from only mild contention. The lock adaptation policy identifies such no-contention locks and configures them to low-latency spin-locks. **qlock**, on the other hand, exhibits considerable lock contention. It is therefore reconfigured into a blocking lock. The improvement in performance of the kernel and the TSP application on using spin locks and reconfigurable locks appear in figure 11 and 12, respectively. (For comparison, the performance graphs with pthreads, and cthreads with blocking locks, have also been included in the figures.) Spinning in Time Warp improves performance by reducing the intervals of trying for locks on which there is relatively little contention, while reconfiguration helps in TSP by making processors ‘back off’ for appropriate intervals in the face of a lot of contention (on **qlock**), thus improving performance.

<sup>10</sup>For brevity, we show the graphs for 10-processor runs of each application. The contention patterns are largely similar for runs with different number of processors.

<sup>11</sup>Although the locks were on a per-processor basis in the Time Warp kernel, (the event lists are distributed across the processors), we report the contention on only one processor. The behavior on the other processors is expected to be similar, the application being symmetric.

## 7 Runtime Configuration – Scheduling for Real-time Time Warp

We are now investigating the runtime configuration of a single thread scheduler – termed scheduler configuration – performing real-time scheduling for Time-Warp discrete event simulations[GFS93b].

While Time Warp optimistic simulation can be seen as increasing dynamic parallelism through speculation, one often needs to get ‘runaway parallelism’ under control through throttling [DeG91]. Dynamic parallelism, by its very nature, cannot be predicted at compile time. The amount of speculation should be large when there is a high amount of dynamic parallelism, and speculation should be throttled when such parallelism reduces.

Monitoring the amount of dynamic parallelism is relatively straightforward in non-speculative execution: the number of busy processors is very often a good enough index in this respect. However, in Time Warp style of speculative execution, processors may be performing incorrect work rather than being idle. One way to monitor ‘recent’ dynamic parallelism is to monitor the number of events committed in some recent past interval of real-time.

We are in the process of implementing a real-time scheduler for speculative computation based on these ideas. For brevity, the real-time aspects of the scheduler are not included in this paper. Here, we briefly mention how reconfiguration can be achieved in such a scheduler.

We have a ‘throttling window’ on each processor. This window determines the maximum number of processed uncommitted events allowable on any processor. When the number of processed uncommitted events becomes equal to the throttling window, event execution is blocked on that processor. (Note that this immediately provides us with an upper bound on the time required for rollback on any processor: processed uncommitted events being the only ones that can roll back.) The throttling window is maintained at a constant number (say, 1) greater than the ‘recent parallelism’, as determined above.

If the throttling window is too small, a small number of events tend to get rolled back; the number of committed events in a ‘recent past interval of real-time’ increases, and the throttling window is therefore increased the next time around. On the other hand, if the throttling window becomes too large, processors tend to get far ahead of each other, and the number of rollbacks increases. This has the effect of reducing the ‘number of committed events in a ‘recent past interval of real-time’, and the feedback control reduces the throttling window: thus achieving dynamic reconfiguration in the scheduler of such a system..

## 8 Conclusions

We have presented the performance of a user-level threads package on the KSR multiprocessor, and compared its performance with a kernel-level threads package that the multiprocessor provides. The user-level package performs substantially better.

Further, we have discussed mechanisms for reconfiguring the locks of such a lightweight threads package, and how that leads to better performance. We have shown the efficacy of our reconfigu-

ration mechanisms by improving the performance of two applications: a TSP application, and the kernel of a Time Warp simulator.

Finally, we have discussed the performance implications of reconfiguring schedulers in lightweight threads packages statically and dynamically, and presented a scheme to perform such reconfiguration.

## Acknowledgments

The Time Warp kernel was originally implemented on a GP1000 BBN Butterfly by Prof. Richard M. Fujimoto.

## References

- [ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.
- [BALL90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in Proceedings of the 12th ACM Symposium on Operating Systems Principles, Dec. 1989.
- [Bla90] D. Black. Scheduling support for concurrency and parallelism in the mach operating systems. *IEEE Computer Magazine*, 23(5):35–43, May 1990.
- [BLL88] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.
- [CC89] Houssine Chetto and Maryline Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, pages 1261–1269, October 1989.
- [CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical report, Computer Science, Carnegie-Mellon University, CMU-CS-88-154, June 1988.
- [CMS93] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (dsa) on large-scale multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-93-25, May 1993.
- [DeG91] Doug DeGroot. Throttling speculative computation: Issues and problems. *Parallel Computing 1991*, pages 19–37, 1991.
- [Fuj89] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1989.
- [Fuj90] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):23–28, January 1990.
- [GFS93a] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. A testbed for optimistic execution of real-time simulations. *IEEE Workshop on Parallel and Distributed Real-Time Systems*, April 1993.

- [GFS93b] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. Time warp simulation in time constrained systems. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, May 1993.
- [Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [LKAS93] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-93/22, Atlanta, GA, March 1993. To appear in 1993 International Conference on Network Protocols.
- [MS93a] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *To appear in the proceedings of International Conference on Parallel Processing*, August 1993. Also available as TR# GIT-CC-93/05.
- [MS93b] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *To appear in the proceedings of high performance distributed computing*, July 1993. Also available as TR# GIT-CC-93/17.
- [Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991. TR# GIT-ICS-91/02.
- [OSS90] David M. Ogle, Karsten Schwan, and Richard Snodgrass. The dynamic monitoring of real-time distributed and parallel systems. Technical report, College of Computing, Georgia Institute of Technology, ICS-GIT-90/23, Atlanta, GA 30332, May 1990. To appear in IEEE TSE.
- [SZG91] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Multiprocessor real-time threads. *Operating Systems Review*, 25(4):35–46, Oct. 1991. Also appears in the Jan. 1992 issue of Operating Systems Review.

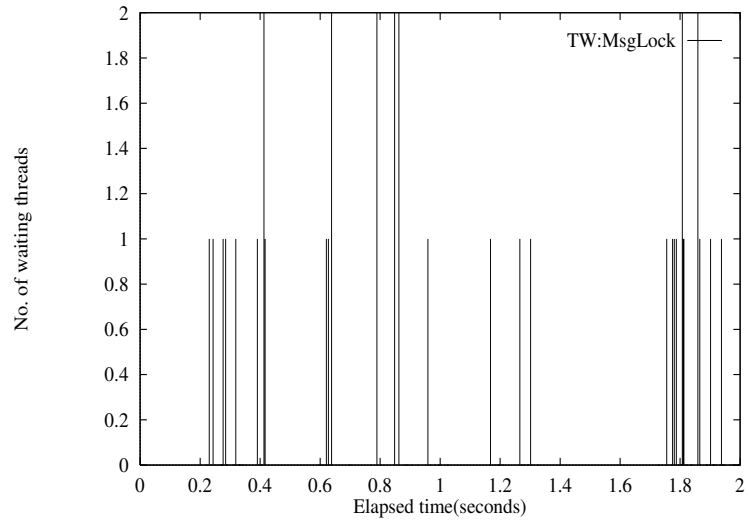


Figure 7: Contention patterns for lock on list of incoming events for Time Warp; 10-processor run

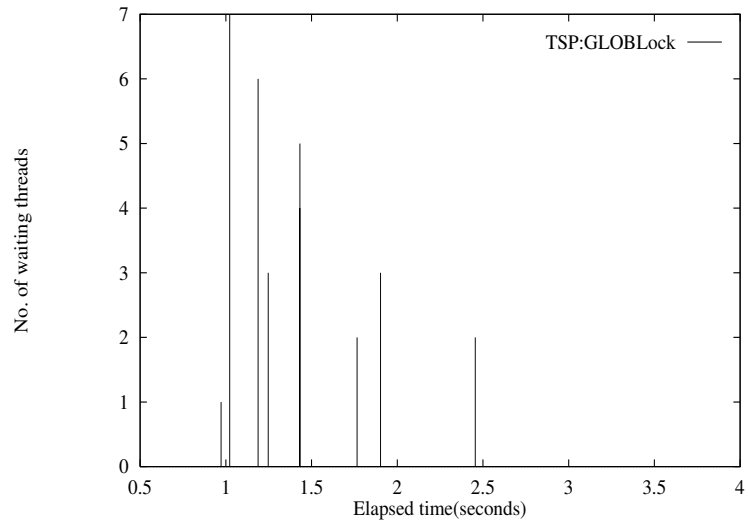


Figure 8: Contention patterns for globlock in TSP; 10-processor run

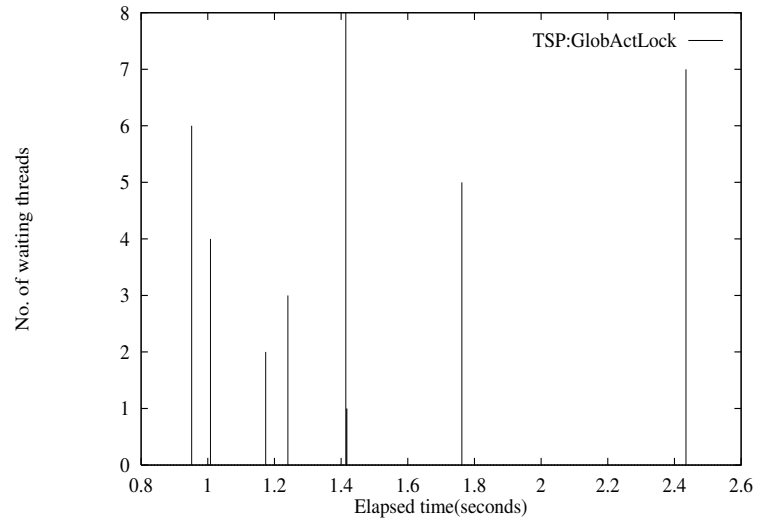


Figure 9: Contention patterns for glob-act-lock in TSP; 10-processor run

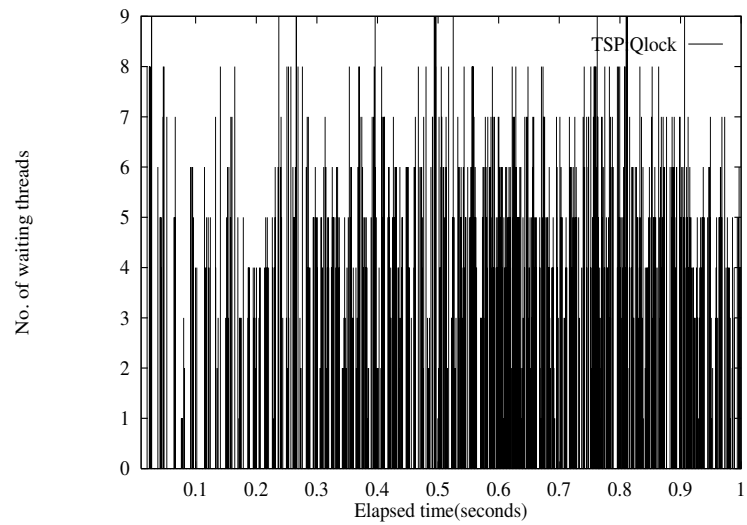


Figure 10: Contention patterns for qlock in TSP; 10-processor run

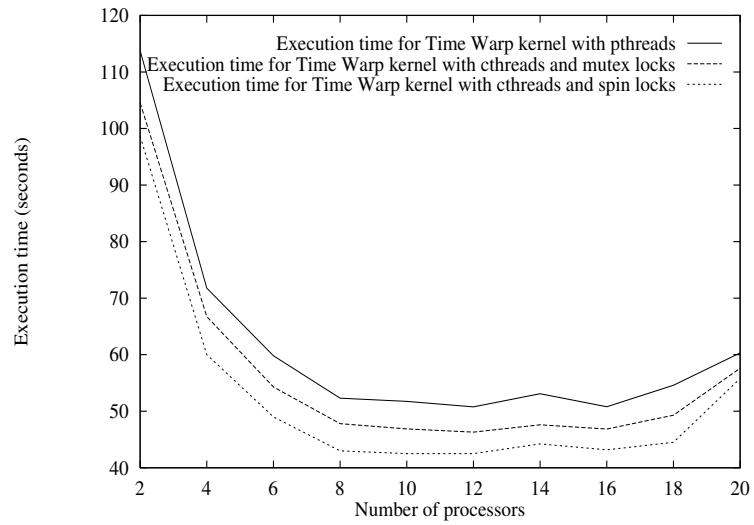


Figure 11: Execution time for Time Warp using cthreads (seconds)

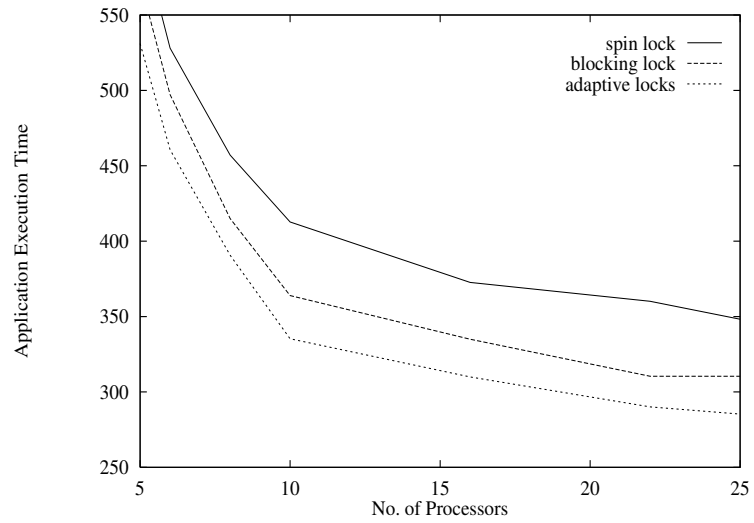


Figure 12: Execution time for TSP using cthreads with various types of locks (seconds)