Experiments with Configurable Locks for Multiprocessors

Bodhisattwa Mukherjee (bodhi@cc.gatech.edu)
Karsten Schwan (schwan@cc.gatech.edu)

GIT–CC–93/05

10 January 1993

Abstract

Operating system kernels typically offer a fixed set of mechanisms and primitives. However, recent research shows that the attainment of high performance for a variety of parallel applications may require the availability of variants of existing primitives or additional low-level mechanisms. One approach to solve this problem is to offer a lightweight, reconfigurable and extensible operating system kernel. An application may configure it to suit its needs, including the selection of appropriate low-level policies, the construction of new primitives on top of the existing ones or the extension with additional primitives. In this paper, we investigate kernel configurability and extensibility for a specific class of operating system primitives: those used for task or thread synchronization. We present an implementation of multiprocessor locks that can be reconfigured statically and dynamically. In addition, we develop a representation for the lock abstraction and an associated reconfiguration mechanism that may be used for the development of other configurable and extensible operating system abstractions.

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

1 Introduction

Past experimentation with parallel machines has demonstrated that the attainment of high performance often requires the customization of operating system mechanisms to each class of application programs. For example, for real-time applications executing on shared memory multiprocessors, the object-based operating system kernels described in [GS89a, SGB87] must offer several representations of objects and object invocations to support the different degrees of coupling, task granularities, and invocation semantics existing in real-time applications[GS93]. Such experiences in the real-time domain are mirrored by work in multiprocessor scheduling[CCLP83, BLL88] that demonstrates the importance of using application-dependent information or algorithms while making scheduling decisions. Similarly, for scientific applications executing on distributed memory machines, the support of multiple semantics of task communication by low-level operating system mechanisms[SB90] or by compiler-generated communication libraries[SBW91] has been shown to enhance application program performance significantly. Lastly, recent research addressing efficient memory models for shared memory[SJG92] and for distributed memory[HA90] machines has made clear that the support of multiple semantics of memory consistency can result in improvements in parallel program efficiency.

This paper explores how an operating system kernel can support application programs in the assembly of program-specific mechanisms and policies[LCC$^+$75]. We address the following questions:

- How can an operating system kernel's abstractions be represented so that they are statically and dynamically configurable?
- What basic mechanisms are required for dynamic kernel configuration?
- Are the runtime costs incurred by dynamic reconfiguration justified by the possible gains of such reconfiguration?

Although it is possible to answer the questions posed above for complex applications and for a variety of operating system constructs[BS91a, BS91b], in this paper, we address these questions for a specific concurrency control construct – the *lock* construct – used for the synchronization of multiple processes in NUMA shared memory parallel application programs. The configurable lock construct described in this paper permits the use of multiple strategies for lock access ranging from 'busy waiting' to 'blocking'. Tradeoffs in the use of these strategies for NUMA machines are demonstrated with measurements on a 32-node BBN Butterfly GP1000 multiprocessor. Some of the experiments described in Section 2 are similar to those performed by Anderson[ALL89] for small-scale UMA machines, but our results are different due to the NUMA characteristics of the BBN Butterfly and of most large-scale parallel machines[IFKR92].

Since the experiments in Section 2 show that different lock configurations result in significant performance differences, Section 3 introduces a dynamically configurable (*reconfigurable*) lock construct. Section 4.1 lists some basic measurements demonstrating the performance penalties due to lock reconfigurability. The application level performance improvements gained from dynamic lock reconfiguration are presented in Section 4.2 using a workload generator. Section 5 compares our work with related research and finally, Section 6 concludes the paper and presents some future directions

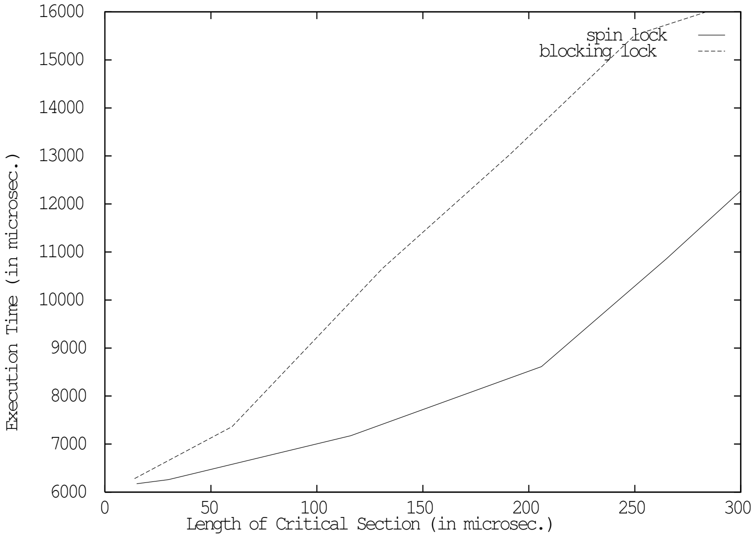

Figure 1: Length of critical section Vs. Application execution time for uniform arrival of lock requests

2 Thread Synchronization with Locks on NUMA Machines

Tradeoffs in program performance due to the use of alternative synchronization constructs have been demonstrated for most parallel architectures, including the experimental C.mmp[WLH81] and Cm*[JS80] multiprocessors, interconnection-network-based machines like the Ultracomputer[Sch80], and bus-based machines like the Sequent Symmetry[ALL89]. For example, in C.mmp users preferred to use simple spin compared to blocking locks offered by the Hydra[WLH81] kernel due to the low latencies associated with spin lock use. More specifically, when concerned with maximizing the speedup of parallel applications, users required a synchronization construct that offered low latency of access to critical sections in response to their availability (regardless of any resulting loads imposed on connections of processors to memory units). In contrast, for bus-based UMA machines, Anderson showed that spin locks can put a significant load on the shared bus, so that efficient use of the parallel machine requires a back-off strategy for spin locks similar to the one used by low-level Ethernet devices[ALL89]. Alternatively, when maximizing processor utilization or when multiple threads exist on each processor, threads accessing critical sections protected by locks should be blocked to enable the execution of other threads performing useful work. Similar conclusions were made by Mellor-Crummey and Scott in [MCS91] while deriving efficient spin-lock implementations to reduce contention on the shared bus.

The measurements in this section study the tradeoffs regarding the use of spin locks vs. blocking locks for critical sections of different lengths and accessed with different frequencies. In contrast to some of the measurements by Anderson on UMA machines [ALL89], we show that NUMA machines behave predictably when spin locks are used. Artificial workloads imposed on the NUMA multiprocessor demonstrate the following characteristics:

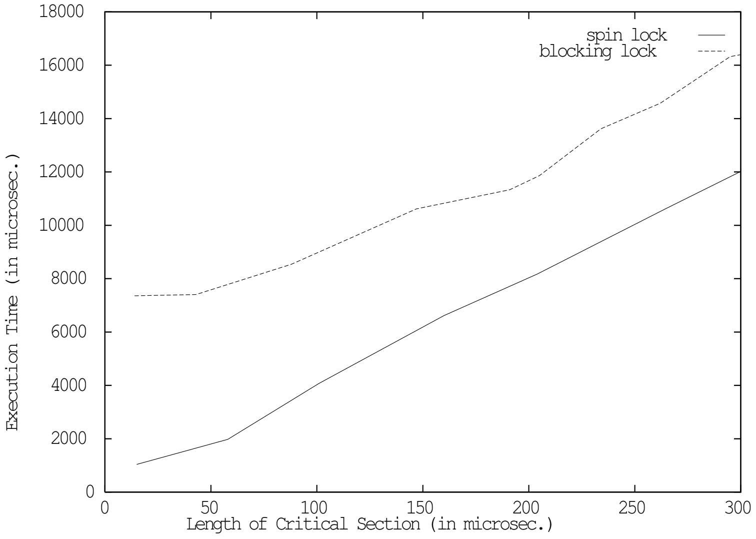

Figure 2: Length of critical section Vs. Application execution time for bursty arrival of lock requests

- Assuming a single thread per processor and a constant frequency of access to critical sections, the execution time of an application program linearly increases with the lengths of critical sections accessed by program's threads. This linear increase is due to increases in the average wait times of threads for critical sections.
- In NUMA machines, spin locks consistently outperform blocking locks when the number of processors exceeds the number of threads. This is due to the reduced latencies of critical section access for spin vs. blocking locks.

These results are demonstrated in Figures 1 and 2 for both bursty (Figure 2) and uniformly distributed (Figure 1) accesses to critical sections. These measurements were made with a workload simulator on a 68020 based 32 node BBN butterfly multiprocessor. The simulator binds one or more thread to each processor which generate locking requests following a user defined pattern.

However, when multiple threads on each processor are capable of making progress, the use of blocking is indicated even for fairly small critical sections, since spinning prevents the progress of other threads not currently waiting on a critical section. The cross-over point while using blocking vs. spinning for the artificial workloads used in our experimentation corresponds to the additional overheads of blocking on the BBN Butterfly (Figure 3).

The experimental results shown above are not surprising. However, these results do imply that any locking mechanism offered by an operating system kernel should permit the mixed use of spinning, backoff spinning[ALL89], and blocking as waiting strategies, depending on the expected or experienced lengths of critical sections protected by such locks. The design of a lock object offering such multiple waiting strategies is presented in the next section.

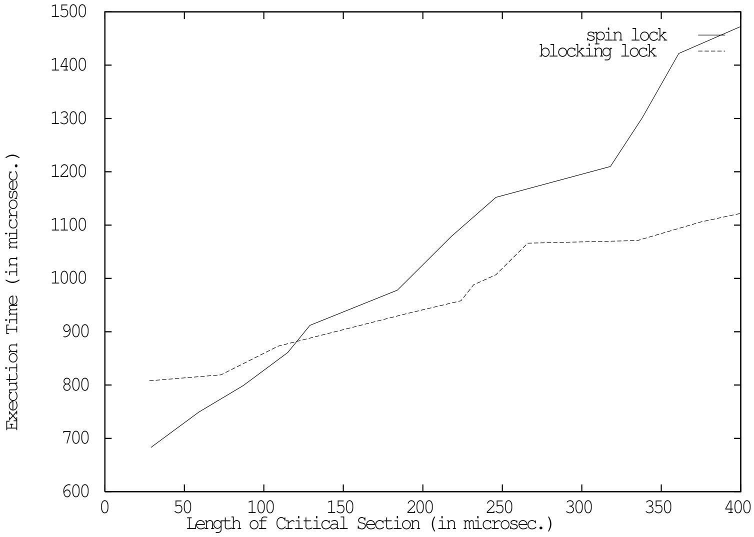

Figure 3: Length of critical section Vs. Execution time of application having useful threads that are capable of making progress

3 Configurable and Reconfigurable Locks

At any specific time a lock can be in one of three different states – locked, unlocked, and idle. A lock is in the "locked" state when a specific thread owns the lock whereas it is in the "unlocked" state when it is free without any waiting threads. A lock enters the "idle" state when it is free but has one or more waiting threads. Figure 4 shows a typical state transition diagram for a multiprocessor lock.

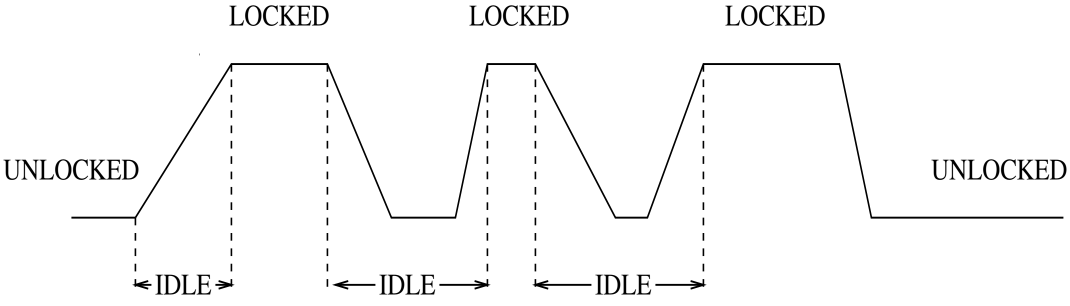

Figure 4: State Transition Diagram of a Lock

An idle state inhibits an application's progress by consuming processor cycles and/or blocking an application thread. A lock becomes idle when the latency of its lock and/or unlock operations are high, when it exhibits an expensive "locking cycle"(an unlock operation followed by a lock operation), and/or when the lock operation interferes with the unlock operation (e.g. a primitive low-level lock is often used to enforce mutual exclusion of a high-level lock data structure. In such a situation, even an unsuccessful lock operation can stop a thread from executing an unlock operation). A spin lock supports lock and unlock operations with minimum latency for locking cycle whereas a blocking lock usually has a higher locking cycle latency. Hence, a spin lock has the least idle state. However, a spin lock may result in increased bus and/or memory contention which may degrade the performance of a shared memory parallel program. The aim of dynamic lock reconfiguration is to reduce the idle time of a lock without significantly increasing bus and/or memory contention resulting in improved application performance. Such reconfiguration is application specific and lock reconfiguration policy will depend on the application locking pattern.

3.1 Reconfigurable Locks – The Model.

Any lock construct has two mechanisms and associated policies determining its behavior: (1) its *scheduling* component determines the delay in lock acquisition experienced by the thread, and (2) its *wait* component specifies the manner in which a thread is delayed while attempting to acquire the lock. Scheduling itself may be divided into three components: (a) a *registration* component logging all threads desiring lock access, (b) an *acquisition* component determining the waiting mechanism and policy to be applied to each registered thread (without registration the lock cannot apply different waiting policies to individual threads), and (c) a *release* component that grants new threads access to the lock upon its release.

Lock configuration can concern changes in each of the components mentioned above. For example, as stated earlier, Anderson et al. in [ALL89] explored the performance effects of using the alternative delay mechanisms of spinning vs. backoff spinning. In contrast, researchers in real-time systems have investigated the costs of priority-based and deadline-based dynamic lock access scheduling[MCS91, Mar91, ZSG92] for multiprocessor systems. The contribution of our work is twofold:

1. describing locks such that their waiting and scheduling behaviors can explicitly changed (statically and dynamically), and
2. demonstrating the usefulness of explicit dynamic reconfiguration. Specifically, although the mechanism of explicit reconfiguration of locks introduces additional overheads, we show that these overheads are outweighed by performance gains due to reconfigurability.

Reconfiguration concerns the dynamic alteration of components of parallel programs[BS91a]. Such reconfiguration is possible only if the components being changed offer an immutable interface to the remainder of the application program. Our research assumes that the object model[Jon79] of software is used to describe components subject to dynamic change. Specifically, an instance of a lock object is uniquely described by its names and methods, the latter implementing the object's functionality. Lock objects are used by invocation of their methods, where both the semantics of invocation and the representation of objects may be class-specific. Namely, each method has some internal *implementation* that may range from being *passive* (i.e., the method's code is executed by the invoker), to *active* threads executed on one or multiple processors[GS93, SGB87] asynchronously to the invoker. Each object also contains global centrally stored or distributed[SB90], static and dynamic internal state accessible to all methods as well as state local to each method. For example, a passive lock object may be described as follows:

```
CLASS passive-lock is

        STATE internal_state <immutable> IS
                queue registration-queue;
                thread-id owner;
                ...
        END

        STATE configurable_attributes <mutable> IS
                int spin-time;
                int delay-time;
                int sleep-time;
                int timeout;
                ...
        END

        OPERATION registration(..);
        OPERATION acquire(..);
        OPERATION release(..);
        OPERATION possess(..);
        OPERATION configure(..);
        .
        .

BEGIN
        Initialization ..
END
```

The object model used above has been shown to be sufficient for representation of a wide variety of parallel application programs on both shared memory and distributed memory machines[GS93, Jon79, GS89a]. However, for reconfiguration and for attainment of high performance, application programs must be aware of additional object properties. These properties may be represented as object *attributes* that may be specified and changed orthogonally to the object's class determined by its methods, as shown by Gheith[GS93] for multiprocessor real-time applications involving attributes like execution *deadlines* on methods or *forward recovery* designations on object classes. We adopt the attribute-based specification originated by Gheith, but we investigate a different class of dynamically changeable attributes. Specifically, we are concerned with object attributes that characterize an object's internal implementation (e.g., the use of spinning vs. blocking in the waiting component of a lock object), and we focus on making dynamic changes to selected implementation attributes. For example, a lock object belonging to the above mentioned `passive-lock` class implements a primitive spin lock when the `sleep-time` is zero and the `spin-time` is infinite. On the other hand, it implements a pure blocking lock when the

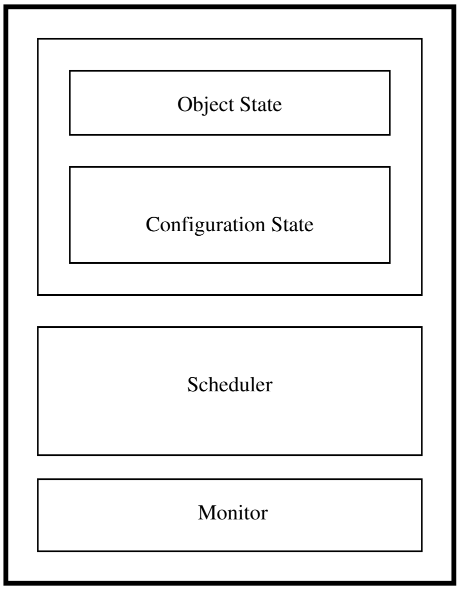

Figure 5: Structure of Lock objects

`spin-time` is zero.

In contrast to Gheith[GS93] and similar to the restricted definition of the object model used by Bihari for on-line adaptation of real-time programs [BS91a], 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, but 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 such as thread waiting queues[SRVO88]. Since object mutability is subject to change over time, the implementation of a reconfigurable lock object presented below possesses an internal *policy* controlling the object's reconfiguration. This policy makes use of object state describing its *ownership* by invokers. Such ownership may be acquired implicitly as part of the invocation of some object method or explicitly by execution of a 'possess' method associated with the object. For example, ownership of the object attribute `spin-time` or `block-time` is acquired implicitly by a thread when it acquires the lock. Also, an external agent (typically, a thread monitoring the state of the lock) may request ownership of an attribute to reconfigure the lock to a desired configuration as shown by the following code segment.

```
passive-lock.possess (a-attribute)
passive-lock.configure (a-attribute, new-config)
```

spin-time	delay-time	sleep-time	timeout	resulting lock
n	0	0	0	pure spin
n	n	0	0	spin (backoff)
0	0	n	0	pure sleep
x	x	x	n	conditional sleep/spin
n	n	n	x	mixed sleep/spin

Table 1: Lock Parameters (`n` = an arbitrary number, `x` = "do not care")

3.2 Reconfigurable locks – Implementation.

As stated above, locks may be reconfigured asynchronously by invocation of a reconfiguration method or synchronously in conjunction with lock or unlock requests. For reconfigurable locks, we have chosen to implement asynchronous lock reconfiguration, in order to avoid forcing all threads that use locks to contain knowledge about lock usage, lock performance, or application program performance. The specific reconfigurable lock presented in this section is implemented such that all lock requests are required to carry an additional parameter based on which each request is directed to the appropriate methods for lock waiting and scheduling. We call such additional parameters *attributes* since they are not required by the methods implementing the object's basic functionality. The attribute used by the reconfigurable lock is the requesting thread's identifier ('thread-id'). This identifier is processed by the lock object's *policy* upon lock invocation as part of the *registration phase* of lock scheduling. The policy next performs *lock acquisition*, which implements a mapping of 'thread id' to the appropriate methods for waiting on the lock, and it also selects the appropriate lock scheduling method for delaying lock access. Both mappings may be changed by reconfiguration operations performed on the lock object described later. As a result, a reconfigurable lock's internal representation contains the following additional information (Figure 5):

- object state (e.g., current lock state, current lock owner, registration information, etc.)
- configuration state (e.g., timeout and spin-time parameters, list of wait methods, etc.), which is shown in detail in Table 1. This Table lists a few such parameters, the possible values for those parameters and the resulting locks. These parameters implement a spectrum of locks as shown in Figure 6. Configuration state not shown in the Table includes architecture-specific information like lock location, object implementation (distributed or centralized objects) etc.

Given the lock implementation outlined above, each lock access request involves the following steps:

Lock: A locking operation consists of the following steps:

1. A requesting thread registers itself with the lock object. At this time, attribute information like thread-id, priorities, ownership, *etc.* is processed by the lock's policy. The overhead of policy execution depends on the number of attributes processed and the complexity of the processing being performed (e.g., a somewhat complex lock scheduling algorithm is described in [ZSG92] for real-time locks). The registration overhead in the configurable lock implementation is the cost of one write operation on primary memory[2]. As shown in Section 4.2, application performance gain due to dynamic lock reconfiguration easily compensates for such registration costs.

---

[2] registration of the thread identity

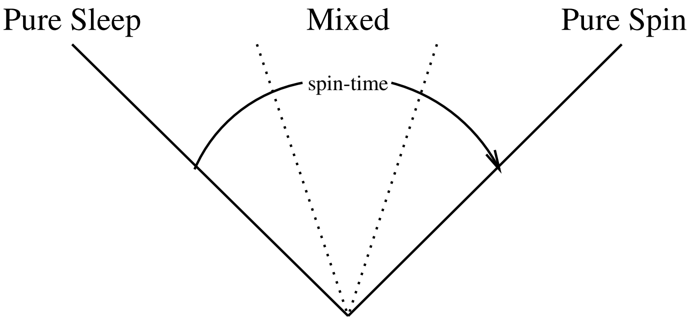

Figure 6: A Spectrum of Multiprocessor Locks

2. If lock status indicates that the thread must wait for the lock, then the waiting method is determined by the acquisition module of the lock object. Current implementation of configurable lock maps requests to methods for spinning, blocking, backoff spinning, conditional locking, and advisory locking.

Unlock: An unlock operation consists of the following steps:

1. The last part of the lock objects's policy is the release module, which selects the next thread that is granted access to the lock.
2. The release module's selection (scheduling) policy may consist of a simple access to a thread-id noting the next thread to be executed (as in handoff scheduling[Bla90a]) or it may execute more complex scheduling strategies.

The reconfigurable lock object also contains a monitor module which senses or probes user-defined parameters. This module implements a user-controlled lightweight thread monitoring system. The information gathered from the monitor can be used by the internal reconfiguration policy and/or an external agent (possibly another application thread) to determine the state of the lock which, in turn, is used to decide on a new lock configuration. The resident policy which is responsible for the lock reconfiguration is called as the lock adaptation policy. Such adaptation policy depends on the locking pattern. Implementations of thread monitoring system[GS92] and adaptable locks[MS93] are described elsewhere.

4 Performance Evaluation

4.1 Formal Characterization

Let V_i be a state variable with value v_i in domain D_i ($v_i \in D_i$). Let SV be the set of variables that constitute the state of an object. SV consists of two subsets IV (set of variables in internal state) and CV (set of variables in configuration state).

$SV = IV \cup CV$ where,
$IV = \{V_1, V_2, \ldots, V_n\}$ and

$$CV = \{U_1, U_2, \ldots, U_n\}$$

The actual values of the variables in CV determine the waiting policy for a lock. Let CV_i be an instance of the sub-state CV of the object, i.e.

$$CV_i = \{x_i \mid U_i = x_i \wedge U_i \in CV \wedge x_i \in D_i\}$$

Then, the set of waiting policies of a lock object can be represented as

$$\Phi = \{CV_i \mid CV_i \texttt{ is an instance } \text{of } CV\}$$

Let Γ be the set of lock schedulers. As mentioned earlier, a lock scheduler consists of three functions (registration, acquisition, and release). Hence, an instance the set Γ is an ordered triple and can be expressed as

$$\Gamma_i = \{\Gamma_i^{Reg}, \Gamma_i^{Acq}, \Gamma_i^{Rel}\}$$

The set C of possible lock configurations is:

$$C = \Gamma \times \Phi$$

The operations defined on lock objects are:

1. Υ : Υ is a state transition operation and is formally expressed by a variation of axiomatic rules:

   $$\boxed{SV_{pre} : \Upsilon : SV_{post}\ [t]}$$

   Where, SV_{pre} and SV_{post} refer to the object states before and after the operation respectively. Υ modifies only the internal state of an object, therefore, can be more precisely defined as:

   $\Upsilon : IV_i \rightarrow IV_j$
   $\Upsilon = \Upsilon_l \vee \Upsilon_u$, where
   Υ_l is the lock operation and can be expressed as:

   Υ_l: $\Gamma^{Reg}; \Gamma^{Acq}$

   Υ_u is the unlock operation and can be expressed as:

   Υ_u: Γ^{Rel}

   t specifies the cost of the specified operation and is expressed in terms of number of memory reads and writes :

   $t = n_1Rn_2W$, where $n_1 > 0 \wedge n_2 > 0$

2. Ψ : Ψ is a reconfiguration operation and is formally expressed as:

   $$\boxed{C_{pre} : \Psi : C_{post}\ [t]}$$

   Where, C_{pre} and C_{post} refer to the object configurations before and after a reconfiguration operation respectively. A configuration C_i is a tuple $\langle \Gamma_i, \Phi_i \rangle$ where $\Gamma_i \in \Gamma$ and $\Phi_i \in \Phi$. Ψ refers to the requested configuration action and is expressed as:

   $\Psi : \langle \Gamma_i, \Phi_i \rangle \rightarrow \langle \Gamma_j, \Phi_j \rangle$

As mentioned above, t specifies the cost of the configuration operation and is expressed in terms of number of memory reads and writes.

A simple dynamic alteration of waiting mechanism of a lock needs only one memory read and one memory write.

$\langle$mutex, X$\rangle$: Ψ^{spin}_{mutex} : $\langle$spin,X$\rangle$ [1R1W]
$\langle$spin, X$\rangle$: Ψ^{mutex}_{spin} : $\langle$mutex,X$\rangle$ [1R1W]
$\vdots$

However, alteration of scheduler is more expensive. It requires three memory writes for three submodules, one memory write to set a flag (to implement the configuration delay), and another memory write to reset the flag (when all the pre-registered threads are served, the old scheduler is discarded).

$\langle$X, fifo$\rangle$: $\Psi^{priority}_{fifo}$: $\langle$X,priority$\rangle$ [1R5W]
$\langle$X, priority$\rangle$: $\Psi^{handoff}_{priority}$: $\langle$X,handoff$\rangle$ [1R5W]
$\vdots$

A complex reconfiguration of a lock happens by a collection of the above operations. The cost of such a reconfiguration is easily obtained by adding costs of the individual operations.

3. I : I is an initialization operation and is defined as:

   $I : IV_i \cup CV_i \cup \Gamma_i \rightarrow IV_0 \cup CV_0 \cup \Gamma_0$ where,

   IV_0, CV_0, and Γ_0 are the initial values of IV, CV, and Γ respectively.

4.2 Costs of Object Reconfiguration

This section describes the basic costs of non-configurable lock implementations and compares them with the costs of the operations provided by the reconfigurable lock object used in our research. The following measurements are taken on a 32-node BBN Butterfly GP1000 NUMA multiprocessor using a multiprocessor version of Cthreads as the basis[Muk91, SFG+91].

Table 2 lists the latencies of the lock operations for different lock implementations available on the BBN multiprocessor (provided by the hardware, operating system and the Cthreads library). A "local lock" refers to a lock which is located in the local physical memory whereas a "remote lock" is located in a non-local memory module. The `atomior`[1] function, which implements a low level atomic or operation (similar to `test-and-set`), is used to implement various locks. The *spin-with-backoff* lock is a variation of the backoff spin lock suggested by Anderson et al.[ALL89]. A thread requesting ownership of such a lock spins once, and if the lock is busy, waits (back offs) for an amount of time proportional to the number of active threads waiting for the processor. As expected, the primitive spin-lock has minimum latency, whereas the blocking-lock exhibits a maximum. The latency of the configurable lock is comparable to that of a primitive spin lock because a lock operation for configurable locks initially spins for the lock before deciding to block the requesting thread.

The costs of unlock operations for various lock implementations are listed in Table 3. The spin locks implement unlock operations with minimum latency, whereas, the blocking lock, as expected, has the

---

[1] provided by the BBN Butterfly multiprocessor hardware

Lock type	local lock (micro seconds)	remote lock (micro seconds)
atomior[1]	30.73	33.86
spin-lock	40.79	41.10
spin-with-backoff	40.79	41.15
blocking-lock	88.59	91.73
configurable lock	40.79	41.17

Table 2: Cost of the Lock operation for different locks

Lock type	local lock (micro seconds)	remote lock (micro seconds)
spin-lock	4.99	7.23
spin-with-backoff	5.01	7.25
blocking-lock	62.32	73.45
configurable lock	50.07	61.69

Table 3: Cost of the Unlock operation for different locks

highest latency. The latency for the configurable lock exceeds that of spin locks due to the extra work required to check for currently blocked threads. As shown by some experiments in the next section, this extra cost is compensated by the performance gain due to reconfiguration.

A thread waits for a busy lock until the current lock owner releases the lock. As mentioned earlier, the cost of a locking cycle (an unlock followed by a lock operation) on a busy (locked) lock determines the duration of the "idle state" of the lock. Table 4 lists the costs of the locking cycle for some static implementations of locks. A *spin-with-backoff* lock has an expensive locking cycle due to the "backoff" cost whereas the "blocking" cost adds to the locking cycle of a pure blocking lock. As shown in Table 5, a configurable lock has the least expensive locking cycle when configured as a spin lock and has the most expensive locking cycle when configured as a blocking lock. The cost of the locking cycle of a configurable lock, configured as a combination of spin and block, lies between these extremes.

Dynamic configuration (also known as reconfiguration) occurs at run time. The owner of a lock or an external agent possessing an attribute of the object may choose to alter its configuration. Dynamic configuration changes pose a few problems which are best stated by the following question:

Lock type	local lock (micro seconds)	remote lock (micro seconds)
Spin	45.13	47.89
Spin-with-backoff	320.36	356.95
Blocking-lock	510.55	563.79

Table 4: Cost of successive Unlock and Lock operation on an already "locked" lock

Configured as	local lock (micro seconds)	remote lock (micro seconds)
Spin	90.21	101.38
Blocking	565.16	625.63

Table 5: Cost of successive Unlock and Lock operation on an already "locked" configurable lock

Operation	local lock (micro seconds)	remote lock (micro seconds)
possess	30.75	33.92
configure(waiting policy)	9.87	14.45
configure(scheduler)	12.51	20.83

Table 6: Cost of Lock Configuration Operations

Once the configuration changes, what happens to the already registered threads? There are two solutions to this problem. If the waiting policy changes, the first solution traverses the registration queue and alters the waiting policies of all the registered threads. If the scheduling policy changes, it moves all the registered threads from the old queue to the new one. The second solution does not change the configuration of the lock until all the pre-registered threads are served. The first solution is not scalable because the length of the registration queue does not have an upper bound. Hence, our implementation incorporates the second solution. The configuration action does not take effect immediately. Such a delay is referred to as a configuration delay. The effect of such delay on reconfiguration operations is part of our future work and is not discussed in this paper

Table 6 lists the costs of the basic dynamic configuration operations. As explained earlier, the "possess" operation is used by an external agent[1] to acquire exclusive ownership of a lock attribute. This operation is rarely used since reconfiguration, in most cases, is done by the lock owner. The cost of this operation is comparable to a primitive *test-and-set* operation. As shown, scheduler reconfiguration is more expensive than waiting policy reconfiguration. Simple dynamic configuration of the waiting policy requires only one memory read and one memory write whereas, alteration of the scheduler requires three memory writes for three submodules, one memory write to set a flag (to implement the configuration delay), and another memory write to reset the flag (when all the pre-registered threads are served, the old scheduler is discarded). However, the configuration costs listed in Table 6 do not capture total configuration delay.

4.3 Experiments with a Reconfigurable Object

In this section, we list the results of five experiments performed with reconfigurable locks. The previous section shows that a reconfigurable lock exhibits a higher locking cycle latency compared to primitive spin locks. The experiments in this section demonstrate that application performance gains due to

---

[1] A thread or a process which is not the current owner of the lock

dynamic reconfiguration outweigh such performance penalties.

The first experiment, which compares the performance of three locks obtained by changing the scheduler configuration, demonstrates the use of application specific lock schedulers. The second experiment changes some attributes of reconfigurable locks to effect different waiting policies. In this experiment, we show that the optimal waiting policy for a lock depends on the application's locking pattern. Since such patterns are not generally known, lock policies can clearly be improved by dynamic reconfiguration. The third experiment demonstrates the application of a configurable lock (called speculative or advisory lock) for variable length critical sections.

The fourth and the fifth experiments demonstrate the utility of implementation specific configurations. The fourth experiment compares application performance using centralized spin locks vs. distributed spin locks whereas the final experiment compares application performance using passive and active locks respectively. All experiments are performed with a workload generator on a 32 node BBN butterfly multiprocessor using a multiprocessor version of Cthreads as the basis.

4.3.1 Scheduler Configuration

As mentioned earlier, a lock scheduler schedules the incoming requests to a critical section. The first interesting result appearing in this section compares the performance of the following three lock configurations using a common class of multiprocessor applications, applications structured as client-server programs. The experiment demonstrates improved application performance when priority or handoff locks are used in place of FCFS locks.

FCFS and Priority lock: While FCFS lock scheduling is most common in multiprocessor lock implementations, non-preemptive-priority locks can be used in applications exhibiting specific locking patterns for improved application performance. Use of priority locks has been widely discussed in the real-time domain[Mar91]. Such locks can also be useful in client-server models of computation. Specially, when a server is flooded with requests from many clients, the priority of its threads may be raised so that a server thread can acquire the locks for the critical sections, it shares with the clients (*e.g.,* message buffers) faster, thereby resulting in faster service. FCFS locks are fair but priority locks are inherently unfair

Handoff lock: A handoff lock scheduler takes hints to select the next thread to be assigned to the critical section. The releasing thread hands off the critical section directly to the selected thread. Handoff locks, like priority locks, are useful in a client-server model of computation. A handoff lock does not guarantee fairness and is typically application specific[1].

In this experiment, one thread (executing on a dedicated processor) is designated to be a server thread serving many client threads. Communication between server and clients is performed via shared message buffers. A client thread enqueues a request to the server thread and waits for a reply on the shared

---

[1] A user must have a clear understanding of the application's behavior, especially the locking patterns, to use a handoff lock policy efficiently

FCFS lock (micro seconds)	Priority lock (micro seconds)	Handoff lock (micro seconds)	Performance Gain
463937.5	-	403735.69	13%
463937.5	419879.49	-	9.5%

Table 7: Performance of Lock Schedulers

buffer. Each of the above lock configurations are used in the experiment to implement the shared buffer. Table 7 compares the resulting program performance. The priority and the handoff locks perform better than the FCFS lock because they allow the server to make progress in preference to clients, therefore serving the clients at a faster rate. The priority lock performs better than the handoff lock in this case due to the extra overhead required by the latter to accept user hints.

Priority locks can be implemented in more than one way. Two straightforward implementations are stated below:

1. The first implementation implements a priority queue for the waiting threads. The release module of the lock scheduler always selects the thread with the maximum priority.
2. The second implementation assigns a priority to the lock object which is used as a threshold of priorities for the requesting threads. Only threads with higher priorities than the threshold are eligible for the lock. The lock is assigned in an FCFS fashion among all eligible threads.

This experiment uses the second implementation of priority locks. Whenever the server thread is flooded with many requests, the lock priority is dynamically altered to temporarily raise the threshold priority above client priority thereby making clients ineligible for the locks.

This experiment clearly demonstrates the utility of application specific lock schedulers. For improved performance an application requires lock schedulers most appropriate for its requirements (locking patterns exhibited by the threads of the application). Note that we are not aware of any current lock implementations (provided by operating system or libraries) which allows users to alter lock schedulers (as supported by reconfigurable locks) to suit their requirements.

4.3.2 Waiting Policy Configuration

As explained earlier, the internal state of a lock object contains attributes that decide whether a waiting thread spins, sleeps or does both. Changing these parameters, statically or dynamically, changes the waiting policy of a requesting thread. A few sample lock configurations obtained by varying the parameters of its internal state are spin, blocking, and combined locks. The second experiment compares the performance of these three configurations using a uniform locking pattern.

Spin lock: If the sleep time is zero (or the spin time is infinite) and the spin time is nonzero, a thread spins while waiting for the lock.

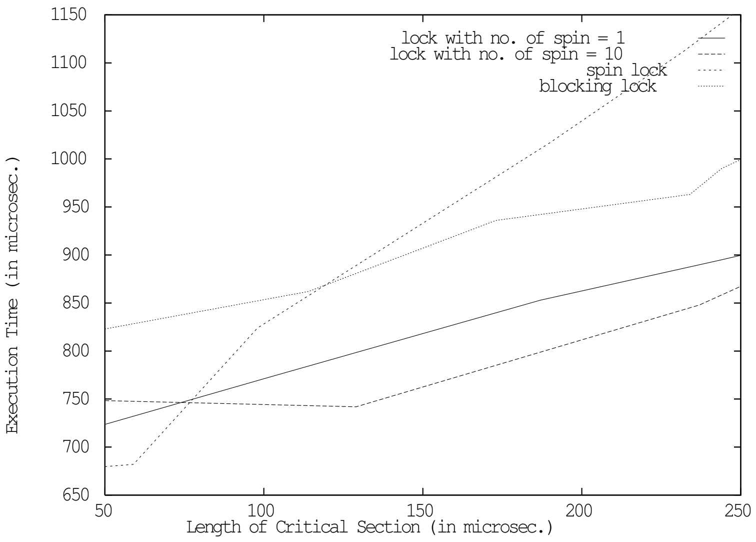

Figure 7: Length of critical section Vs. Application execution time

Blocking lock: If the spin time is zero and the sleep time is nonzero, a waiting thread directly goes to sleep until awakened by a thread (possibly a releasing thread) or a timeout signal. The lock scheduler wakes up a specific thread or all the sleeping threads depending on the release policy.

Combined lock: A combined lock results when both spin time and sleep time are set to nonzero. A thread spins as well as sleeps while waiting. The resulting waiting policy is decided by the actual values of the parameters. In a typical waiting policy, a thread initially spins for a certain time and if the lock is still busy, goes to sleep unless awakened by someone. In another waiting policy for combined locks, a thread spins and sleeps in turn until it acquires the lock.

The second experiment repeats the experiment done in Section 2 on performance of spin vs. blocking when multiple threads on each processor are capable of making progress (Figure 3) using combined locks with the latter waiting policy. Figure 7 shows the results of the experiment, and compares the performance of combined locks with spin and blocking locks. Due to the low latency, spin locks outperform others when the critical section is small. However, for larger critical sections, the figure shows a distinct performance advantage in favor of combined locks.

The figure shows the performance of two combined locks – one that spins 10 times initially before blocking, another that spins once before blocking. In this experiment the former outperforms the latter for larger critical sections, However in general, the optimal number of initial spins of combined locks will depend on various application characteristics such as its locking pattern, length of critical sections *etc.* Furthermore, the optimal waiting policy for a lock will be different during different phases of a computation from which we hypothesize that a waiting policy based on dynamic feedback (reporting the state of a lock) is essential for better application performance. This topic is addressed in our

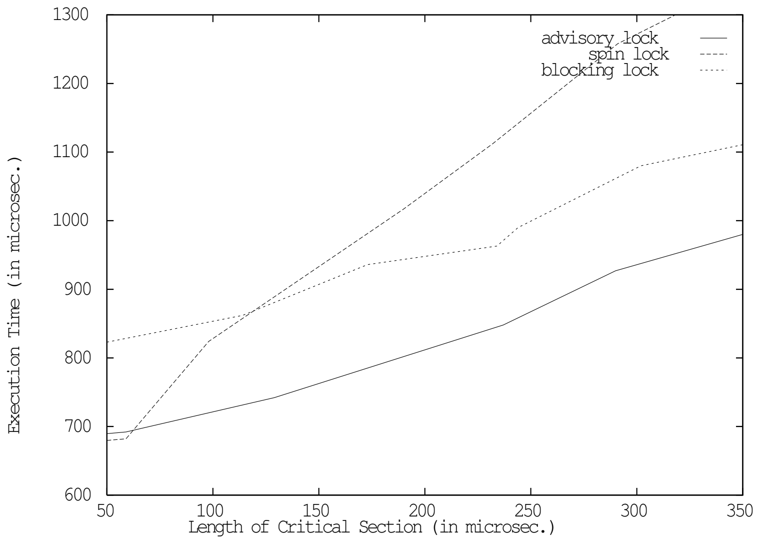

Figure 8: Length of critical section Vs. Application execution time

future research, where we use lock monitor information to identify the current lock state and vary lock configurations, thereby resulting in speculative dynamically reconfigurable locks.

The third experiment studies the performance of another lock configuration called advisory locks. In this experiment, we study the effect of changing the waiting policy in different phases of a computation (as suggested by the result of the last experiment). This experiment demonstrates considerable performance improvement due to such changes over a non-configurable spin or blocking locks.

Advisory/Speculative lock: As the name suggests, the owner of such a lock advises other requesting threads whether to spin or sleep while waiting by (dynamically) changing the parameters of its internal state. Advisory locks are a result of direct application of configurable locks. In general, the length of each synchronization lock tenure may vary significantly in different phases of computation. The current lock owner is the best source of information for the length of lock ownership. Hence, the current owner of a synchronization lock advises or configures the lock for a requesting thread. The owner also may choose to change the advice at different stages in the critical section. Two such situations are listed below:

- A critical section may have multiple conditional paths of varying lengths. The owner may change the lock to be a sleeping lock if it takes a long path inside the critical section or may change the lock to spin if it takes a short path.
- If a critical section is long, the owner initially sets the lock to be a sleeping lock. When it reaches the end of the critical section, it changes the lock to spin.

This experiment uses an application performing critical sections of varying lengths. As the length of the critical sections increases, the owner thread alters the configuration of the lock. Figure 8 compares

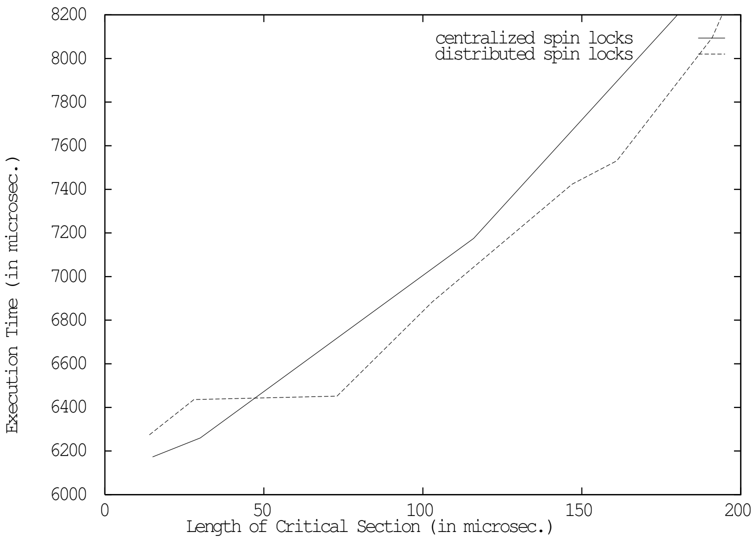

Figure 9: Length of the critical section Vs. Application time

the performance of such a lock with simple spin and blocking locks. As the figure shows, advisory locks outperform ordinary spin or blocking locks for variable length critical sections. Since the remaining length of a critical section owned by a thread decreases with time, the figure also suggests that the advice be changed at regular intervals dynamically by the owner to reflect the remaining length of the tenure for improved performance.

4.3.3 Implementation Specific Configuration

The implementation of a lock may be altered using architectural hints. The fourth experiment compares application performance using centralized spin locks vs. distributed spin locks. The results of this experiment is not too surprising. However, they do support the conclusions made by Anderson et al.[ALL89] that distributed implementation of locks, especially in NUMA machines, often improves application performance to a certain extent, however small it may be.

Centralized Vs. Distributed or Semi-consistent locks: A Centralized implementation does not replicate a lock in different processors. Waiting for centralized locks may require considerable number of remote memory accesses which are expensive and may cause switch/bus contention. On the other hand, a distributed implementation of a lock replicates the lock in more than one processors, thereby minimizing remote memory accesses. The replicas need not be consistent always as long as the implementation guarantees a fair mutual exclusion.

The result of the experiment performed using three processors is shown in Figure 9. Although the figure demonstrates a small performance advantage in favor of distributed locks, we hypothesize that

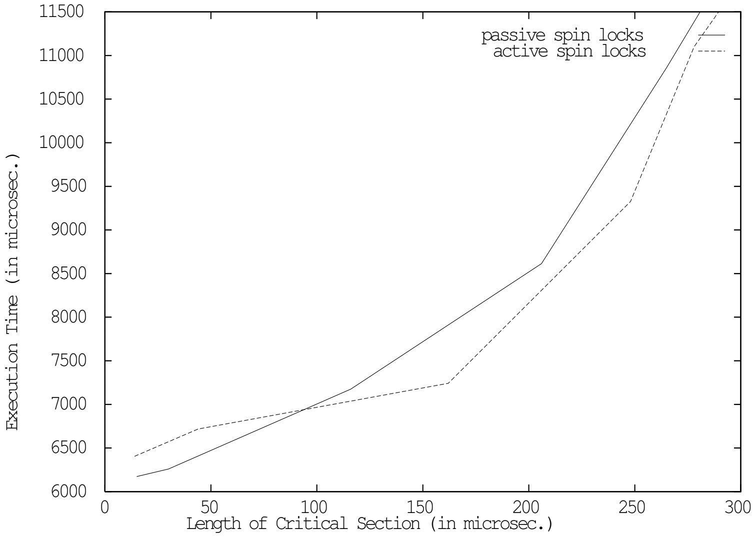

Figure 10: Length of the critical section Vs. Application time

improvement will be more when the number of processors are large. Similar results have already been shown by others[ALL89], hence they are not included in this paper.

The final experiment compares two alternative lock implementations – passive and active locks.

Passive Vs. Active locks: If a lock object has a permanent thread bound to it, we refer to it as an active lock. The thread is bound to the lock at the lock creation time and it remains bound through out its lifetime. Figure 10 compares the performance of an application using active vs. passive spin locks. The experiment demonstrates that active locks are little cheaper because it takes the responsibility of executing the release module from the owner processor, thus providing the releasing processor more time to execute useful application-specific code. However, applications using active locks need more number of processors to execute.

We have also implemented a few common lock configurations to demonstrate the generality of the structure of reconfigurable lock objects. The remaining portion of this section lists these lock configurations obtained using reconfigurable lock objects.

A *read-write* lock is implemented using a scheduler that allows multiple reader threads inside a critical section. Such a lock scheduler implements an algorithm similar to a multiprocessor scheduling algorithm with some constraints. A read-write scheduler can be combined with a priority or a handoff scheduler to create variants where readers have priority over writers or vice versa.

Recursive locks[1] are implemented by maintaining the lock-ownership information in the registration module. An attempt to re-acquire the same lock is easily detected because a lock object knows the identity of its owner. Recursive locks are more expensive than the normal locks because each requesting

---

[1] recursive locks are implemented in MACH [Bla90b] kernel to eliminate the lock-reentrancy problem.

thread performs an extra memory write at registration time.

Simple *conditional locks* result when the timeout parameter is set accordingly. A thread waiting for such a lock returns unsuccessfully if it cannot acquire the lock in a specified time.

5 Related Research

Some of the notions introduced in PRESTO[BLLW88, BLL88], CHOICES[CJR87] and CHAOS[GS89a, GS93, GS89b, SGZ90b, SGZ90a] are somewhat similar to this work. PRESTO supports two kinds of synchronization primitives: spinlocks and synchronization objects. A synchronization object is made of a spinlock (spinlock implements the mutual exclusion), a queue (threads block in this queue), and a few objects required to implement its semantics. Structurally, PRESTO's synchronization object is somewhat similar to reconfigurable locks. However, a synchronization object does not support dynamic attribute reconfiguration and object state monitoring. CHOICES is an example of an object based reconfigurable operating system which can be tailored for a particular hardware configuration or for a particular application. The focus of CHOICES is to structure the operating system kernel in an object oriented way whereas, the focus of our research is to build a configurable operating system kernel at the thread level. Even though we use the object model at the application level, we do not use objects to build the run-time system. CHAOS[2] is a family of object-based real-time operating system kernels. The family is *customizable* in which existing kernel abstractions and functions can be modified easily. As opposed to CHAOS objects, a reconfigurable lock contains its own mutable attributes and an internal configuration policy to guide any reconfiguration operation.

There has been a lot of work on multiprocessor synchronization. In [ALL89], Anderson et al. compare the performance of a number of software spin-waiting algorithms. They propose a few efficient spin-waiting algorithms such as *Ethernet style backoff* algorithm (introducing delay between successive spins analogous to Ethernet's backoff or Aloha), *software queueing* of spinning processors etc. We have built similar backoff spin locks as different configurations of the reconfigurable lock. In [MCS91], Mellor-Crummey et al. propose a new scalable algorithm (a list-based queuing lock, also known as *MCS* lock) which generates O(1) remote references per lock acquisition, independent of the number of processors attempting to acquire the lock. We have built a similar lock (distributed lock) as a configuration (implementation dependent configuration) of the reconfigurable lock.

6 Conclusion and Future Work

The contribution of our work is twofold: First, we, propose a structure for lock objects such that their waiting and scheduling behaviors are easily changed (statically or dynamically), and then we demonstrate the usefulness of dynamic reconfiguration of such lock objects.

The experiments in Section 2 imply that an operating system kernel should provide a locking mecha-

---

[2] A **C**oncurrent, **H**ierarchical, **A**daptable **O**perating **S**ystem supporting atomic, real-time computations.

nism that permit the use of different waiting and request handling strategies depending on the expected or experienced lengths of critical sections. Even though the presented reconfigurable lock model exhibits some performance penalties compared to primitive locks (as shown in Section 4.1), the experiments in Section 4.2 shows that in most cases, these penalties are outweighed by the performance gain due to reconfigurability.

In this paper, we show that it is essential to have application specific lock schedulers for increased performance (as priority schedulers and handoff schedulers are used in some classes of applications). We also show that the dynamic change of waiting policy attributes results in improved application performance.

Section 4.2 demonstrates that the optimal waiting policy of a lock is a function of number of spins, blocks, and/or timeouts. This function depends on various application characteristics such as its locking pattern, length of its critical sections *etc.* Our future research focuses in part on finding the optimal waiting policy for an application specific lock by speculative dynamic reconfiguration. Next, we will extend this idea to build self-adaptable objects. Such an object uses a builtin monitor and an adaptation algorithm to implement a feedback loop to configure its own attributes. We will use the concepts of reconfiguration and adaptation in other operating system components as well to build a lightweight reconfigurable operating system kernel.

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.

[Bla90a] D. Black. Scheduling support for concurrency and parallelism in the mach operating systems. *IEEE Computer Magazine*, 23(5):35–43, May 1990.

[Bla90b] David. L. Black. *Scheduling and Resource Management Techniques for Multiprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, July 1990. Techreport CMU-CS-90-152.

[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.

[BLLW88] B. Bershad, E. Lazowska, H. Levy, and D. Wagner. An open environment for building parallel programming systems. In *Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 1–9, July 1988.

[BS91a] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[BS91b] T. Bihari and K. Schwan. Dynamic adaptation of real-time software for reliable performance. *ACM Transactions on Computer Systems*, May 1991.

[CCLP83] G. Cox, M. Corwin, K. Lai, and F. Pollack. Interprocess communication and processor dispatching on the intel 432. *ACM Transactions on Computer Systems*, 1(1):45–66, February 1983.

[CJR87] R. Campbell, G. Johnston, and V. Russo. Choices (class hierarchical open interface for custom embedded systems). *Operating Systems Review*, 21(3):9–17, July 1987.

[GS89a] Ahmed Gheith and Karsten Schwan. Chaosart: A predictable real-time kernel. In *Butterfly Users Group Meeting, BBN Advanced Computers INc., Rochester, NY*, April 1989. Talk abstracts do not appear in proceedings.

[GS89b] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106–125, July 1989.

[GS92] Weiming Gu and Karsten Schwan. A monitoring and visualization system for parallel and distributed systems. Technical Report Draft, College of Computing, Georgia Institute of Technology, 1992.

[GS93] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[HA90] Phil W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distrbuted shared memories. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 302–311, 1990.

[IFKR92] H. Burkhardt III, S. Frank, B. Knobe, and J. Rothnie. Overview of the ksr1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[Jon79] A.K. Jones. The object model: A conceptual tool for structuring software. In *Operating Systems - An Advanced Course*, pages 7–16. Springer Verlag, New York, Editors R. Bayer, R.M. Graham and G. Seegmueller, 1979.

[JS80] Anita K. Jones and Peter Schwarz. Experience using multiprocessor systems: A status report. *Surveys of the Assoc. Comput. Mach.*, 12(2):121–166, June 1980.

[LCC$^+$75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the 5th Symposium on Operating System Principles, Austin, Texas.* Assoc. Comput. Mach., SigOps, Nov. 1975.

[Mar91] E. P. Markatos. Multiprocessor synchronization primitives with priorities. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 1–7, May 1991.

[MCS91] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9, 1991.

[MS93] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. Technical Report GIT-CC-93/17, College of Computing, Georgia Institute of Technology, 1993.

[Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[SB90] Karsten Schwan and Win Bo. Topologies – distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

[SBW91] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6), 1991.

[Sch80] J.T. Schwarz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–543, Oct. 1980.

[SFG$^+$91] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A cthread library for multiprocessors. Technical Report GIT-ICS-91/02, College of Computing, Georgia Institute of Technology, 1991.

[SGB87] Karsten Schwan, Prabha Gopinath, and Win Bo. Chaos – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[SGZ90a] K. Schwan, A. Gheith, and H. Zhou. Chaos-arc: A kernel for predictable programs in dynamic real-time systems. In *Seventh IEEE Workshop on Real-Time Operating Systems and Software, Univ. of Virginia, Charlottesville*, pages 11–19, May 1990.

[SGZ90b] Karsten Schwan, Ahmed Gheith, and Hongyi Zhou. From chaos-min to chaos-arc: A family of real-time multiprocessor kernels. In *Proceedings of the Real-Time Systems Symposium, Orlando, Florida*, pages 82–92. IEEE, Dec. 1990.

[SJG92] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, May 19-21 1992.

[SRVO88] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A language and system for parallel programming. *IEEE Transactions on Software Engineering*, 14(4):455–471, April 1988.

[WLH81] William A. Wulf, Roy Levin, and Samuel R. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill Advanced Computer Science Series, 1981.

[ZSG92] Hongyi Zhou, Karsten Schwan, and Ahmed Gheith. Dynamic synchronization of real-time threads for multiprocessor systems. In *Proceedings of the 3rd Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1992.