Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package

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

GIT–CC–93/17

10 February 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 best suited for specific applications. One approach to addressing this need is to offer an adaptable and extensible operating system kernel. In this paper, we present a model for adaptive objects and associated mechanisms which may be used for the development of high performance operating system kernels for parallel and distributed systems. We use the model to implement a class of multiprocessor locks namely, adaptive locks which adapt themselves according to user-provided adaptation policies to suit any application locking pattern. Using a well-known multiprocessor application, the Travelling Sales Person program, we demonstrate the performance advantage of adaptive locks over existing locks.

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

1 Introduction

Past research with parallel and distributed machines has demonstrated that the attainment of high performance requires the customization of operating system mechanisms and policies to suit each class of application programs. For example, for real-time applications executing on shared memory multiprocessors, the object-based operating system kernels described in [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[GS]. 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 it clear that the support of multiple semantics of memory consistency can result in improvements in parallel program efficiency.

This paper explores the support of application programs by kernel-level assembly of program-specific mechanisms and policies[WLH81]. We address the following questions:

- How an operating system kernel's abstractions be represented so that they can detect changes in application requirements and adjust to suit such changes?
- Are the runtime costs incurred by some dynamic adaptation justified by the possible performance gains?
- What basic mechanisms are required for dynamic kernel adaptation?

In this paper, we present a model for adaptive objects. We use this model to implement a class of multiprocessor locks, adaptive locks. Adaptive lock objects detect changes in application defined lock attributes at runtime, and adjust to such changes by using application specific adaptation policies. In this paper, we demonstrate that the added runtime cost of dynamic adaptation is outweighed by the ensuing performance gains, using measurements on a 32 node BBN Butterfly GP1000 multiprocessor.

The next section lists a few related results from our previous research to motivate the use of adaptive objects. Section 3 introduces the notions of reconfigurable and adaptive objects.

Section 4 demonstrates the utility of such objects for multiprocessor synchronization using three implementations of a well-known multiprocessor application, Travelling Sales Person(TSP) program. The implementation of adaptive locks and some basic measurements demonstrating the performance penalties due to monitoring of lock parameters and reconfigurability are discussed in section 5. Section 6 compares our work with related research and finally, section 7 concludes the paper and presents some future directions.

2 Previous Work

In [MS93], we investigate the advantages of dynamic configuration for a specific class of operating system primitives: those used for task or thread synchronization. Using artificial work-loads on a NUMA multiprocessor, we demonstrate that:

- 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.
- When multiple threads on each processor are capable of making progress, the use of blocking is preferred even for fairly small critical sections, since spinning prevents the progress of other threads not currently waiting on a critical section.

Furthermore, the experiments show that any locking mechanism offered by an operating system kernel should permit the mixed use of spinning, back-off spinning, and blocking as waiting strategies. The design of a reconfigurable lock object is presented in [MS93] offering such multiple waiting strategies.

A reconfigurable lock object permits the explicit alteration (statically and/or dynamically) of its waiting and scheduling[1] behaviors. Experiments with reconfigurable locks with a workload generator on a 32 node BBN Butterfly multiprocessor (using a multiprocessor version of Cthreads[Muk91] as the basis) demonstrate that the application performance gains due to dynamic reconfiguration outweigh any performance penalties caused by the added runtime costs. Furthermore, several commonly held beliefs regarding multiprocessor synchronization are verified:

- For improved performance, an application requires lock schedulers that are most appropriate for the locking patterns exhibited by its threads. One experiment compares the performance of three lock schedulers – FCFS, Priority, and Handoff using a common class of multiprocessor applications: applications structured as client-server programs. For such applications, priority locks exhibit the best performance whereas FCFS locks exhibit the worst, thus demonstrating the importance of application-specific lock schedulers[MS93].

[1] The scheduling component of a lock object determines the delay in lock acquisition experienced by a thread, and consists of three disjoint components – *registration*, *acquisition*, and *release*[MS93]

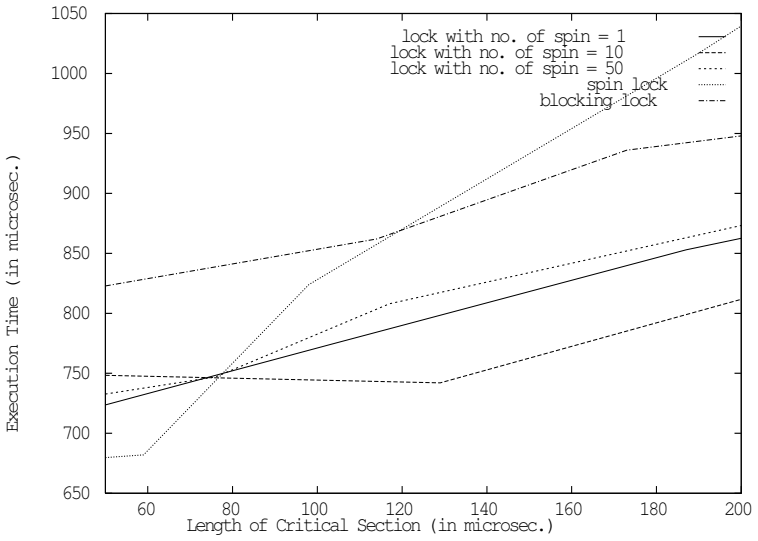

Figure 1: Length of critical section Vs. Application execution time

- Second experiment compares the performance of some implementation-specific lock configurations (centralized vs. distributed locks, passive vs. active locks), thereby demonstrating the advantages of changing implementations to re-target such objects to different architectural platforms (*e.g.,* from UMA to NORMA, for different processor partitions, *etc.*).
- The optimal waiting policy for a lock depends on the application's locking pattern and can be improved by dynamic configuration. A third set of experiments alters selected attributes of reconfigurable locks to effect different waiting policies demonstrating the effect of different waiting policies on performance. Furthermore, these experiments suggest that a speculative or advisory lock[2] performs well for variable length critical sections.

We repeat one result of our previous research in Figure 1 which suggests that a waiting policy based on dynamic feedback (reporting the state of an object) is essential for improved application performance. The figure compares the performance of an application using combined locks[3] vs. pure spin and blocking locks. The figure shows the performance of three combined locks – one that spins 10 times initially before blocking, one that spins 50 times before blocking, and one that spins only once before blocking. The results demonstrate that the lock spinning 10 times performs better than that spinning once for certain lengths of critical sections. However, the lock spinning 50 times performs worse than the lock spinning 10 times for critical sections of the same length. The results of this experiment suggest that the optimal number of initial spins of combined locks will depend on various application characteristics such

[2]The owner of such a lock advises other requesting threads whether to spin or sleep while waiting, dynamically changing some attributes of its internal state during different phases of computation[MS93]

[3]A thread waiting for a combined lock spins as well as sleeps while waiting according to a user-defined policy[MS93]

as its locking pattern, length of critical sections *etc.* Furthermore, the optimal waiting policy for a lock might differ during different phases of a computation. These results give credence to the main contribution of this paper. Namely, a waiting policy based on dynamic feedback (reporting the state of an object) should lead to improved performance for a large number of shared memory multiprocessor applications (especially important for applications with unknown lock patterns, applications with frequently changing lock patterns *etc.*). The contribution of our work is twofold:

- presenting a structure for adaptive objects which can be used to build different operating system abstractions for parallel and distributed systems.
- applying the structure to implement multiprocessor locks to investigate the usefulness of adaptability. Specifically, although the mechanisms required for gathering information and for dynamic lock reconfiguration introduce additional overheads, we show that these overheads are outweighed by performance gains due to adaptability using a well-known multiprocessor application – Travelling Sales Person program.

3 Adaptive Objects

In a typical thread-object model of computation, threads are treated as active entities whereas objects are treated as passive entities. Objects are primarily used to encapsulate code and data, whereas computation results from a thread's invocation of a method/procedure defined by an object.

The object model is important to our research because the encapsulation property of objects hide their implementation from other program components, thereby permitting the alteration of an object implementation without changing its interface. PRESTO[BLL88] system and the CHAOS[GS, GS89, SGZ90], use this property of objects to construct configurable parallel programming environments. CHAOS also captures differences in object implementation by introduction of explicit object attributes that may be used for static or dynamic object configuration. With respect to configurability of objects, we classify objects into three types: non-configurable, reconfigurable and adaptive.

Non-configurable objects: Such objects are mainly used to encapsulate code and data. Specifically, an instance of an object is uniquely described by its names and methods, the latter implementing the object's functionality. 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.,

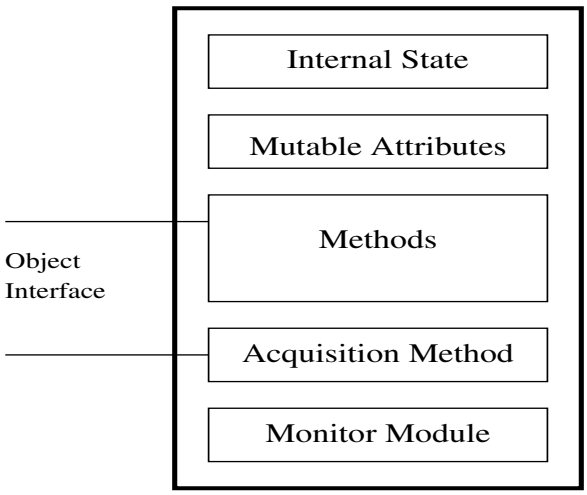

Figure 2: Structure of Reconfigurable Objects

the method's code is executed by the invoker), to *active* threads executed on one or multiple processors[GS, 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.

Reconfigurable objects: Reconfigurable objects provide mechanisms to alter the implementation of one or more of their methods dynamically (at execution time) without changing their interface. Figure 2 illustrates the structure of configurable objects presented in [MS93]. In the figure, "Configurable Methods" include the non-configurable object methods, the configurable object methods (possibly configured using attributes) and the methods implemented by the reconfiguration mechanism.

Reconfiguration concerns the dynamic alteration of components of parallel programs[BS91]. Such reconfiguration is possible only if the components being changed offer an immutable interface to the remainder of the application program. 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. We use an attribute-based specification to investigate a class of dynamically changeable attributes. Specifically, we are concerned with object attributes that characterize an object's internal implementation.

Similar to the restricted definition of the object model used by Bihari for on-line adaptation of real-time programs [BS91], we assume that changes in object attributes may be performed

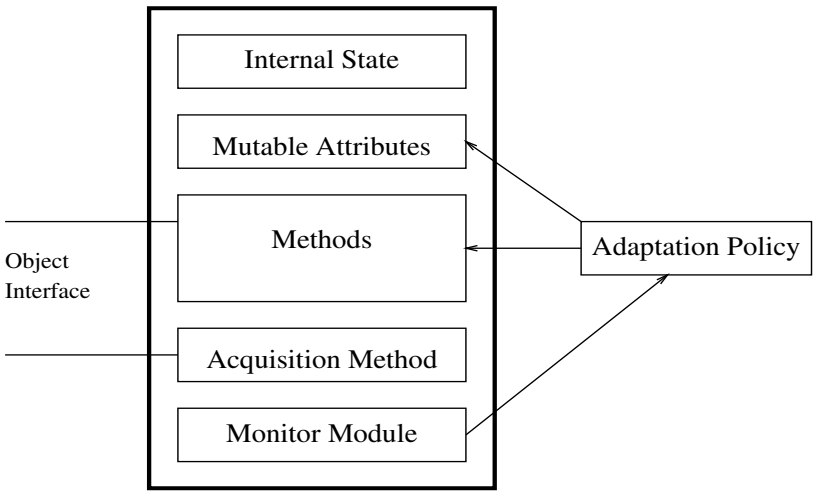

Figure 3: Structure of Adaptive Objects

both synchronously or asynchronously with method invocations[GS]. This requires the introduction of two additional time-dependent properties of object attributes: (1) mutability and (2) ownership. An attribute is *mutable* whenever its current value may be changed. Since attribute mutability is subject to change over time, the implementation of a reconfigurable lock object possesses an internal *mechanism* to enact the object's reconfiguration. This policy makes use of object state describing its *ownership* by invokers. Such ownership of an object attribute is acquired in one of the two ways – implicitly, when an object method belonging to a specific set of methods is invoked, explicitly when the "acquisition" method is invoked by an external agent (typically, a thread monitoring the state of the object)[MS93].

Adaptive objects: In addition to the components of a reconfigurable object, an adaptive object contains built-in monitoring mechanism that provides information about its attributes and a user-provided adaptation policy. The information from the monitor module may be used implicitly or explicitly to tune an object for improved performance. Figure 3 shows the structure of such an object.

The monitor module senses changes in those object characteristics that are required for reconfiguration of the mutable attributes and of the method implementations. The implementation of this module needs to be efficient to avoid extra overhead associated with the adaptation. Section 5 presents an implementation of such a monitoring mechanism. In a configurable object, the monitor information is normally fed to an external agent (possibly a thread belonging to the same application) for interpretation so that the external thread senses the current state

of an object, and dynamically configures it. However, if the adaptation module is included as a part of the object, then the object is referred to as an adaptive object. An adaptive object implements a feedback loop consisting of the monitor mechanism, the adaptation policy, and the reconfiguration mechanism.

We have identified a few issues and tradeoffs regarding adaptive objects which control their performance and are mentioned briefly in the remaining part of this section.

Monitoring Cost vs. Amount of Information: The quality of monitored data depends on two factors:

Diversity Factor: The diversity factor of monitored information captures the range of data monitored. The quality of information improves as the variety of "useful" monitored data increases.

Sampling Rate: The sampling rate specifies the frequency of information monitoring (*i.e.,* how often a state variable is monitored). A higher sampling rate improves the information quality, however, an over-abundance of data causes "information overload".

The overall cost of monitoring object state depends on the cost of the underlying monitor mechanism, the diversity factor, and the sampling rate. As the diversity factor and the sampling rate increase, the quality of adaptation improves but monitoring overhead increases.

Adaptation Cost vs. Quality of Adaptation: The overall cost of adaptation increases as the quality of adaptation decreases which depends on the user-defined adaptation algorithm, diversity factor of the monitored data (how many different data to consider to take a decision), and the sampling rate (how often does the policy execute).

Coupling of the feedback loop: The coupling between the monitor module and the adaptation module, and between the adaptation module and the reconfiguration mechanism affect the performance of adaptive objects. If the adaptation module lags the monitor module by a large amount, overflow of information may cause the adaptation module to make a reconfiguration decision based on a past object state rather than the current state. The adaptation lag becomes even more pronounced if the reconfiguration mechanism also lags the adaptation module. Therefore, to avoid such lags an adaptive object implements a "closely-coupled" feedback loop.

3.1 Formal Characterization

Let V_i be a state variable with value v_i in domain D_i ($v_i \in D_i$). Let IV be the set of variables that constitute the internal state and CV be the set of variables constituting the mutable attributes. Then, the object state SV is represented as:

$SV = IV \cup CV$ 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 an implementation-dependent portion of object configuration. 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 policies specified by object attributes are represented as

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

Let Γ be the set of methods implementing the object interface. Then, the set C of possible object configurations is:

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

The configurable methods consist of:

1. $\Upsilon : \Upsilon$ 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:

   $$\Upsilon : IV_i \rightarrow IV_j$$

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

   $t = n_1 R n_2 W$, where $n_1 > 0 \wedge n_2 > 0$

2. $\Psi : \Psi$ 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 complex reconfiguration of an object happens by a collection of primitive operations. The cost of such a reconfiguration is 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.

The feedback loop is implemented by the monitor module(M), the adaptation policy(P) and the reconfiguration mechanisms:

$$M \stackrel{v_i}{\Longrightarrow} P \stackrel{d_c}{\Longrightarrow} \Psi$$

where, v_i is the value of a state variable and d_c is a reconfiguration decision.

4 Experimentation With TSP Application

An adaptive lock object, used for task or thread synchronization in multiprocessor operating system kernels and/or user-level thread libraries, detects changes in application request (locking) pattern and adapts itself for improved performance. Adaptation concerns altering the waiting and scheduling policies of locks to suit application requirements. Section 5 presents the implementation of such locks in detail. In this section, we discuss the performance advantage of using such locks in TSP application.

A wide class of parallel applications work by construction of search trees from an initial problem (possibly a graph). Branch-and-bound algorithms, which are commonly used in optimization problems, are members of this class. We studied a specific application belonging to the class of branch-and-bound algorithms called the travelling sales person problem.

This specific algorithm is the LMSK branch and bound algorithm for finding the shortest tour of a fully connected graph. The algorithm [SBBG89] proceeds by dynamic construction of a search tree, at the root of which is a description of the initial problem. Independent subproblems are generated by selection of some edges from the graph and creation of children of the root. The algorithm continues to choose leaf nodes and expand them until a tour is found. Once a tour is found for any subproblem, the search space may be pruned by deletion of some unnecessary branches and leaf nodes. When all leaf nodes have been expanded and pruned, the lowest tour is the minimum tour of the graph.

We have used a Thread library[Muk91] to implement the TSP algorithm as a collection of asynchronous cooperating threads[CS93]. 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 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 more node in the work queue.

We implemented the above algorithm in three different ways by varying the implementation of the shared abstractions from centralized to distributed, with and without a load balancing strategy. All implementations use 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. After studying the locking patterns for all the locks, we found that `glob-low-lock` and `globlock` do not suffer from any contention. The lock adaptation policy (as presented in this section) identifies such no-contention locks and configures them to low-latency spin-locks. `qlock` and `glob-low-lock`, on the other hand, exhibit considerable lock contention (shown later in this section for each implementation). As expected, these patterns demonstrate that the distributed implementation (Figure 6) exhibits a much lower lock contention for `qlock` than the centralized implementation (Figure 4).

The customized lock monitor for the adaptive locks, used in the following experiments, senses the number of waiting threads (sampled once during every other unlock operation) to be used by the adaptation module. The adaptation policy used by the adaptation module is very simple and straightforward as given below:

FUNCTION `simple-adapt() <input: no-of-waiting-threads>` IS

IF `no-of-waiting-threads` = 0
    Configure the lock to be pure spin;
ELSE IF `no-of-waiting-threads` $\leq$ *Waiting-Threshold*
    Increase no-of-spins by `n`;
ELSE IF `no-of-waiting-threads` $>$ *Waiting-Threshold*
    Decrease no-of-spins by `2*n`;

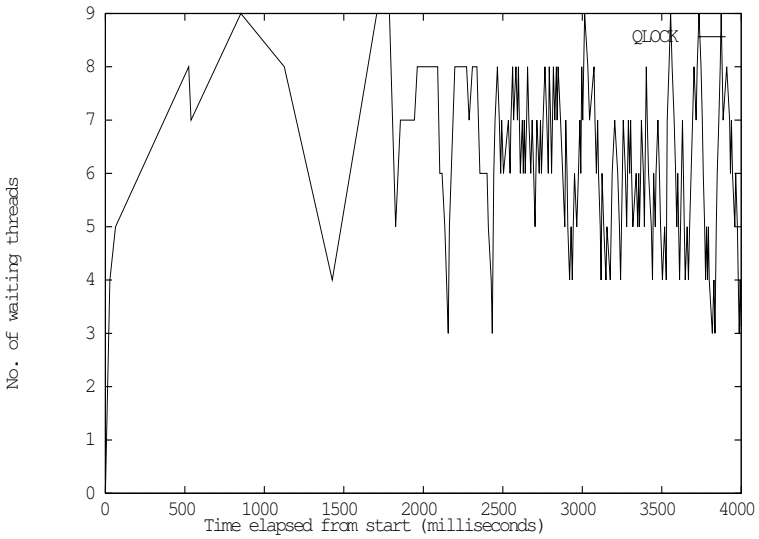

Figure 4: Locking Pattern for "QLOCK" in the Centralized Implementation

IF `no-of-spins` ≤ 0
Configure the lock to be pure blocking;

END

In the above algorithm, *Waiting-Threshold* specifies the threshold of the number of waiting threads for a lock which determines when the number of spins should be raised and `n` is a lock-specific constant. The threshold and `n` are different for different locks and depend on the locking pattern and the length of the critical section. The constants *Waiting-Threshold* and `n` need to be varied to get the optimized adaptation policy for a specific lock. The next step of our research focusses on finding the exact relationship between *Waiting-Threshold*, `n`, and the length of the protected critical section.

The remaining part of this section briefly discusses each implementation focussing on the application locking patterns, and the effect of adaptive locks on the performance of these implementations. The measurements shown in this section are taken using 10 processors with one thread per processor (each searcher thread executes on a dedicated processor) on a BBN Butterfly NUMA multiprocessor for a problem of size 32 (*i.e.,* the number of cities of the initial TSP problem is 32).

Sequential (milliseconds)	Blocking Lock (milliseconds)	Adaptive Lock (milliseconds)	Percentage Improvement
20666	3207	2636	17.8%

Table 1: Performance of the Centralized Implementation using blocking lock vs. adaptive lock

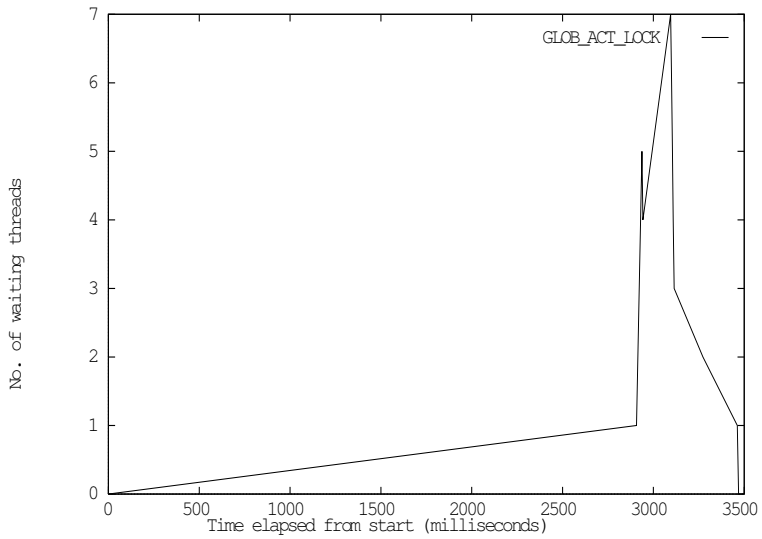

Figure 5: Locking Pattern for "GLOB-ACT-LOCK" in the Centralized Implementation

Centralized Implementation: This implementation uses a global centralized (located on one physical node) tour value and a global centralized work queue. As a result, the best tour value and the work sharing queue are consistent (global ordering is strictly maintained in the work sharing queue) and pruning of the search-space tree is optimal[CS93]. However, as shown in Figure 4 and Figure 5, the lock contention (especially for `qlock`) is high for both the locks.

Table 1 compares the execution time of the centralized parallel implementation using blocking locks running on 10 processors with the sequential implementation showing a 6.5 times speedup. However, when the blocking locks were replaced by adaptive locks (using the simple adaptation policy listed above), the same application shows up-to 17.8% performance improvement over the centralized implementation using blocking lock. This performance gain is attributed to two factors: Firstly, the adaptive lock identifies the no-contention locks and alter them to the lowest-latency lock, and secondly, the adaptation policy varies the waiting policy by varying the spin time and blocking time depending on the monitor input to decrease the

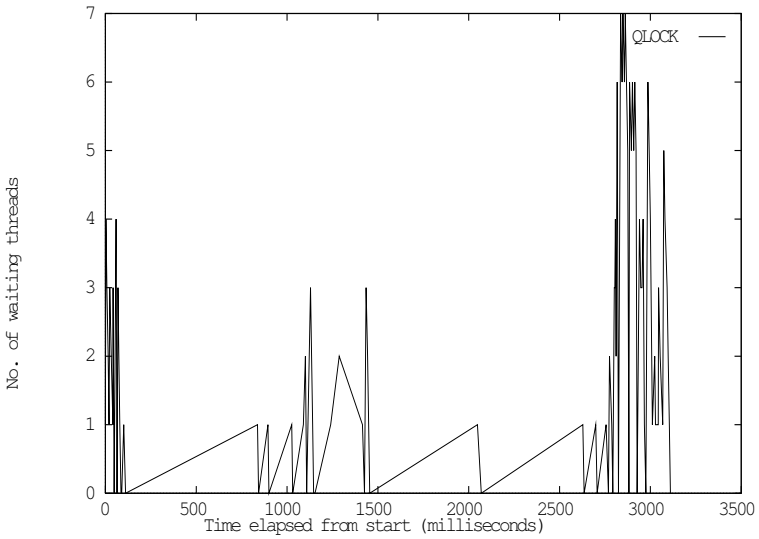

Figure 6: Locking Pattern for "QLOCK" in the Distributed Implementation

average waiting time of each thread.

Distributed Implementation without Load balancing: The distributed implementation of TSP uses a distributed work sharing queue and a distributed best tour value. Each processor maintains a local work queue; the local queues are connected by a ring. When the local queue is empty, a searcher thread gets work from the next non-empty remote queue along the ring. Similarly, each processor maintains a local copy of the tour value and once a new best tour is found, propagates it to the rest of the processors. As a consequence, the distributed implementation may suffer from useless node expansions due to an inconsistent tour value and partially ordered work sharing queues. However, as shown by Figure 6 and 7, lock contention is lower compared to that of the centralized implementation.

Table2 compares the performance of this implementation using blocking lock vs. adaptive locks, and demonstrates up-to 12.7% performance improvement in favor of the latter. The distributed implementation performs better than the centralized implementation because most of the work is performed locally reducing the number of remote memory accesses.

Distributed Implementation with Load Balancing of Work Queues: This implementation uses distributed work sharing queue and best tour value with a specific load balancing strategy among the local work queues to improve the global ordering. Each time a searcher

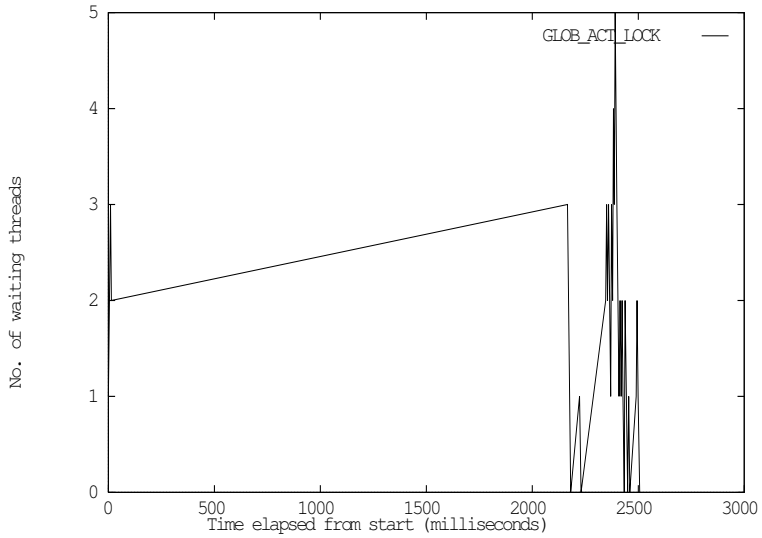

Figure 7: Locking Pattern for "GLOB-ACT-LOCK" in the Distributed Implementation

Blocking Lock (milliseconds)	Adaptive Lock (milliseconds)	Percentage Improvement
2973	2596	12.7%

Table 2: Performance of the Distributed Implementation using blocking lock vs. adaptive lock

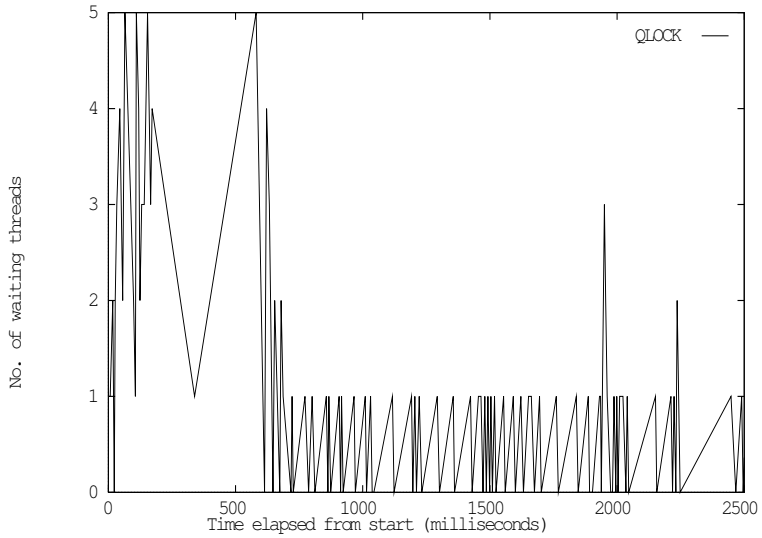

Figure 8: Locking Pattern for "QLOCK" in the Distributed Implementation(with Load Balancing)

Blocking Lock (milliseconds)	Adaptive Lock (milliseconds)	Percentage Improvement
2054	1921	6.5%

Table 3: Performance of the Distributed Implementation (with load balancing) using blocking lock vs. adaptive lock

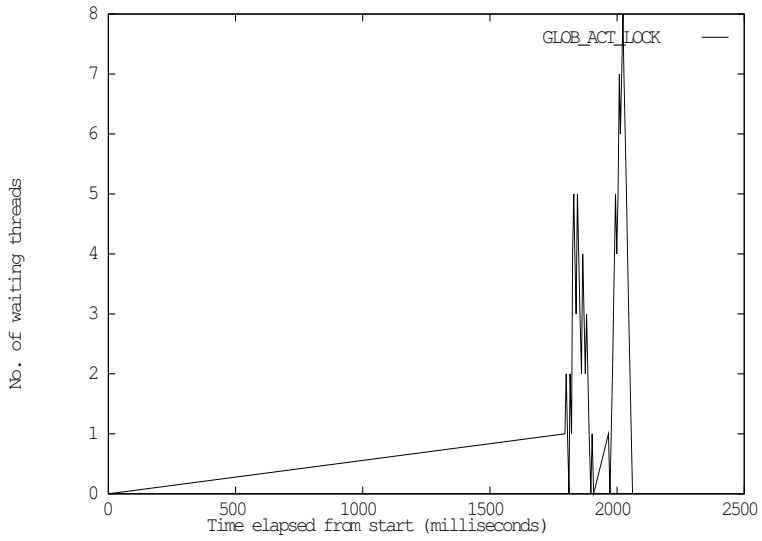

Figure 9: Locking Pattern for "GLOB-ACT-LOCK" in the Distributed Implementation (with Load Balancing)

gets a node, it gets a sub-problem from the queue of the next processor, puts this sub-problem into the local queue, and gets the best sub-problem from the local queue. As expected, lock contention for this implementation (Figure 8 and 9) is lower than the centralized implementation.

Table 3 compares the performance of this implementation using blocking lock vs. adaptive locks showing a 6.5% performance advantage in favor of adaptive locks. The performance advantage is low compared to the centralized version because of reduced synchronization activities.

The results in this section demonstrate that the locking patterns are different for different locks and for different applications. It is important to take such patterns under consideration with other related factors (*e.g.*, length of critical sections, assignment of threads to processors) to determine the waiting policy for a lock. The experiments with TSP showed the utility of

"closely-coupled" adaptation in synchronization. The performance improvement is significant for both the centralized and distributed implementations. Since, the TSP application was programmed using one thread per processor, this performance enhancement is attributed mainly to finding the most suitable waiting policy for a lock (causing a minimum locking cycle). For massively parallel applications we expect the gain to be even higher because the effect of blocking vs. spinning (useful processing vs. wasted processor cycles) is more pronounced. This topic will be addressed in our future research, where we will study a massively parallel application to see the effect of adaptive locks.

5 Adaptive Locks

We used the adaptive object structure to implement a few operating system abstractions on a 32 node BBN Butterfly GP1000 multiprocessor. This section first describes the model and the implementation of adaptive multiprocessor locks, then lists some basic measurements demonstrating the performance penalties due to lock adaptability.

5.1 Implementation

An adaptive lock object, used for task or thread synchronization in multiprocessor operating system kernels and/or user-level thread libraries, senses changes in application request (locking) pattern and adapts itself for improved performance. The structure of such locks are quite similar to that of reconfigurable locks presented in [MS93] and consists of the following components:

Internal state: current lock state, current lock owner, registration information, *etc.*

Mutable attributes: specify the waiting policy of the lock (the policy that governs the manner in which a thread is delayed while attempting to acquire the lock). Table 5.1 lists a few such attributes, the possible values for those attributes and the resulting locks for a simple adaptive lock.

Configurable methods: Lock and Unlock operations are reconfigured by alteration of the lock scheduling policy. Each configurable lock *scheduling* component determines the delay in lock acquisition experienced by the thread and may be divided into three sub-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.

Monitor module: A monitor module implementing a customized lock monitor to sense application locking patterns.

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

Adaptation policy: A user-provided adaptation policy that takes input from the monitor module, and alters the mutable attributes and the scheduling sub-components for improved performance. A simple adaptation policy for the locks in TSP application is presented in Section 4.

The lock monitor module is a customized monitor system obtained from a general purpose thread monitor[GS93] which provides users with a mechanism to insert data collecting sensors and probes into the target application program. The thread monitor implements a local monitor using a monitor thread on a dedicated processor which receives trace data from application threads, performs some low-level processing if necessary and sends them to a central monitor (possibly running in a remote machine) or to the adaptation module. We found such an implementation to be too loosely coupled to be used in adaptive lock objects. Therefore, the customized lock monitor uses the application threads (executing the object methods) to perform information collection, thus making the monitor module closely coupled with the adaptation module.

Lock Parameters[4]

According to the structure of a lock object outlined above a simple adaptive lock may be described as follows:

```
CLASS adapt-lock is

    STATE internal_state <immutable> IS
        queue registration-queue;
        thread-id owner;
        int no-of-waiting-threads;
        ...
    END

    STATE mutable_attributes IS
        int spin-time;
```

[4] n represents an arbitrary number, and x represents the "do not care" condition

```
            int delay-time;
            int sleep-time;
            int timeout;
            ...
        END

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

        MONITOR mon-specs IS
            sensor no-of-waiting-threads <sampling rate:  n>
            ...
        END

        ADAPTATION adapt-policy IS
            void simple-adaptation() <input:  no-of-waiting-threads>;
            ...
        END

BEGIN
    Initialization ..
END
```

Given the lock implementation outlined above, each lock access request involves the following steps[MS93]:

Lock: A locking operation consists of the following steps:

registration: 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 (the "registration" component of the lock scheduler).

acquisition: 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. As shown in Table 5.1, a simple implementation of adaptive lock maps requests to methods for spinning, blocking, back-off spinning, conditional locking, and advisory locking depending on the current state of the object and the application locking pattern.

Lock type	local lock (micro seconds)	remote lock (micro seconds)
atomior[5]	30.73	33.86
spin-lock	40.79	41.10
spin-with-backoff	40.79	41.15
blocking-lock	88.59	91.73
adaptive lock	40.79	41.17

Table 4: Cost of the Lock operation for different locks

Unlock: An unlock operation consists of the following step:

release: The last part of the lock object's policy is the release module, which selects the next thread that is granted access to the lock. 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[Bla90]) or it may execute more complex scheduling strategies depending on the current configuration.

5.2 Performance Evaluation

This section describes the basic costs of non-adaptive lock implementations and compares them with the costs of the operations provided by the adaptive lock object. A formal characterization of reconfiguration operations and their costs is presented in [MS93]. The following measurements are taken on a 32-node BBN Butterfly GP1000 NUMA multiprocessor using a multiprocessor version of Cthreads as the basis[Muk91].

Table 4 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`[6] 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 adaptive lock is comparable to that of a primitive spin lock because a lock operation for adaptive locks initially spins for the lock before deciding to block the requesting thread.

[6] provided by the BBN Butterfly multiprocessor hardware

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
adaptive lock	50.07	61.69

Table 5: Cost of the Unlock operation for different locks

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

The costs of unlock operations for various lock implementations are listed in Table 5. The spin locks implement unlock operations with minimum latency, whereas, the blocking lock, as expected, has the highest latency. The latency for the adaptive lock exceeds that of spin locks due to the extra work required to check for currently blocked threads.

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[MS93]. Table 6 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 7, an adaptive 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 an adaptive lock, configured as a combination of spin and block, lies between these extremes.

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

Table 7: Cost of successive Unlock and Lock operation on an already "locked" adaptive lock

Operation	local lock (micro seconds)	remote lock (micro seconds)
acquisition	30.75	33.92
configure(waiting policy)	9.87	14.45
configure(scheduler)	12.51	20.83
monitor (one state variable))	66.03	-

Table 8: Cost of Lock Configuration Operations

Table 8 lists the costs of the basic mechanisms associated with adaptation. As explained earlier, the "acquisition" method is used by an external agent[7] 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 8 do not capture total configuration delay[MS93].

6 Related Research

Some of the notions introduced in PRESTO[BLL88], CHOICES[CJR87] and CHAOS[GS, GS89, SGZ90] are close to this work. PRESTO uses the encapsulation property of objects to build a configurable parallel programming environment. Structurally, PRESTO's synchronization object is somewhat similar to adaptive locks. However, a synchronization object does not support dynamic attribute reconfiguration, adaptation, 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[8] 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

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

[8] A **C**oncurrent, **H**ierarchical, **A**daptable **O**perating **S**ystem supporting **a**tomic, **r**eal-time **c**omputations.

opposed to CHAOS objects, an adaptive object contains its own mutable attributes, an internal user-provided adaptation policy, and a monitoring mechanism to aid any adaptation decision.

7 Conclusion and Future Work

The contribution of our work is twofold: First, we propose a model for adaptive objects which can be used to build different operating system abstractions for parallel and distributed systems. Then, we applied the model to implement adaptive multiprocessor locks, and demonstrated their utility using various implementations (ranging from centralized to distributed) of the TSP application. We showed that "closely-coupled" adaptations can be used in operating system kernels for improved performance.

In this paper, we show that locking patterns are different for different locks and for different applications. For high performance applications and operating system kernels, it is essential to take these patterns into consideration to build an efficient waiting policy for a particular lock. Our future research focuses in part on studying the effect of applying "closely-coupled" adaptation to alter lock schedulers in different phases of a computation.

Next, we will study a massively parallel application to see the effect of adaptive locks. Finally, we will use the concept of "closely-coupled" adaptation in other operating system components as well to construct an adaptable operating system kernel for parallel and distributed systems to satisfy application requirements.

Acknowledgements

We would like to thank Christian Clemencon for implementing the TSP application using the Thread library on BBN Butterfly multiprocessors. We would also like to thank Kaushik Ghosh for his valuable comments and suggestions on our results.

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.

[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.

[BS91] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, 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.

[CS93] Christian Clemencon and Karsten Schwan. Shared memory vs. fragmented objects for representing shared abstractions in numa multiprocessors: experimentation with a parallel tsp program. Technical Report Draft, College of Computing, Georgia Institute of Technology, 1993.

[GS] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *To appear in ACM Transactions on Computer Systems.*

[GS89] 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.

[GS93] Weiming Gu and Karsten Schwan. A monitoring and visualization system for parallel and distributed systems. Technical Report GIT-CC-93/11, College of Computing, Georgia Institute of Technology, 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.

[MS93] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. Technical Report GIT-CC-93/05, 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. Techreport no: GIT-ICS-91/02.

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

[SBBG89] Karsten Schwan, Ben Blake, Win Bo, and John Gawkowski. Global data and control in multicomputers: Operating system primitives and experimentation with a parallel branch-and-bound algorithm. *Concurrency: Practice and Experience, Wiley and Sons*, pages 191–218, Dec. 1989.

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

[SGZ90] 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.

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