KTK: Kernel Support for Configurable Objects and Invocations

Ahmed Gheith (gheith@austin.ibm.com)
Bodhisattwa Mukherjee (bodhi@cc.gatech.edu)
Dilma Silva (dilma@cc.gatech.edu)
Karsten Schwan (schwan@cc.gatech.edu)

GIT–CC–94/11

28 February 1994

Abstract

The *Kernel Tool Kit* (KTK) is an object-based operating system kernel and parallel programming library that offers explicit support for on- and off-line program configuration. Specifically, KTK allows the specification of *attributes* for object classes, object instances, state variables, operations and object invocations. Attributes are interpreted by *policy* classes that may be varied separately from the abstractions with which they are associated. They can be used to vary object internal implementation and semantics without affecting the methods being invoked.

In this paper, the runtime configuration of KTK attributes is shown to improve the runtime performance of multiprocessor applications. KTK is layered on a portable and configurable parallel programming substrate, a Mach Cthreads compatible runtime library.

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

1 Configuring parallel systems for performance

It is well-known that the performance of a parallel program can be significantly affected by the operating system primitives it is using. Mismatches between a program's desired operating system functionality or state and the primitives, policies, and state offered by the system can cause substantial performance degradation. Such mismatches can arise at the time of program initiation or even during program execution, since a program's pattern of operating system usage can change over time.

Synchronization and scheduling are two well-studied examples of dynamic changes in program behavior involving operating system abstractions. Concerning synchronization, it has been shown that long busy-waits on a single lock can cause severe performance degradation due to memory or switch contention on shared memory multiprocessors. Since the size of a critical section protected by the lock and the contention regarding critical section is often subject to unpredictable change during execution, lock implementations should vary dynamically for best performance[22]. Similarly, since system loads cannot be predicted in multi-user systems, the optimal levels of parallelism (ie., scheduling) for each parallel program cannot be statically predicted and should also be changed dynamically[28].

One reaction of the high performance computing community to potential static or dynamic mismatches between programs and operating systems has been to remove operating systems completely from their application programs, as seen by past 'operating systems' used in hypercube machines[8] offering only basic support for message passing. A similar reaction has been to use only those operating system facilities that offer suitable performance at the loss of ease of programming[11, 29]. However, in the recent past, operating system researchers have addressed the issue of matching operating system to application functionality and performance (1) by offering users the ability to write custom policies on minimal, low-level mechanisms[14, 7] or to tailor their programs to different target operating systems or hardware configurations[25], (2) by removing operating system services from kernel to user levels, ultimately leading to current notions of micro-kernels, and (3) by designing new interfaces between the operating system and users so that selected state is efficiently shared and manipulated by both (e.g., consider recent work on scheduler activations[2]). In addition, researchers have investigated novel means of structuring operating system kernels such that (1)-(3) can be performed conveniently and efficiently, in the past using protection-based techniques[7] and now using object-oriented operating system structures[1, 12, 5, 27, 9].

The current status of research in operating system for high performance can be summarized as follows:

- There are several ways to implement operating systems such that their compile-time configuration is possible, including the use of servers, of object-oriented technology for structuring OS kernels, etc.

- Limited on-line configuration is possible by dynamic re-linking of OS kernels, by dynamic server process creation and deletion, and by dynamically re-directing selected system calls.

The goal of our research is to increase the performance of parallel programs beyond what is currently possible, by dynamically and jointly configuring operating systems and parallel application programs[26, 18]. This distinguishes our work from other efforts addressing distributed or real-time systems[16, 4, 13, 17] (including some of our own past work[3, 9]), where the primary concern has been to maintain certain levels of system response or reliability in the presence of uncertain execution environments. In accordance with this goal, the contribution of the Kernel Toolkit (KTK) is *the provision of mechanisms for the dynamic configuration, inclusion, and use of alternative kernel abstractions.*

In the remainder of this paper, we first present the basic configuration mechanisms offered by KTK. In Section 3, we demonstrate the use of these mechanisms for dynamic configuration of an application-level abstraction in a parallel program: a global queue in a parallel branch-and-bound code solving the travelling salesperson problem (TSP). Section 4 provides some implementation details of KTK and of the global queue in order to explain the performance evaluation of on-line configuration of the TSP queue. In Section 5, additional performance gains are attained by on-line configuration of a second abstraction in TSP, a mutex lock protecting the global queue.

2 The kernel toolkit (KTK)

2.1 Configuring KTK programs

The *Kernel Toolkit* (KTK) is an object-based operating system kernel and programming library that offers explicit support for on- and off-line object configuration:

- KTK allows the specification of (configuration) *attributes* for object classes, object instances, state variables, operations and object invocations.
- Attributes are interpreted by system- or programmer-defined *policies*, which may be varied separately from the abstractions with which they are associated. For example, policies and attributes may be used to vary objects' internal implementations without changing their functionalities, or to vary the semantics and implementations of object invocations without affecting the methods being invoked.
- Dynamic configuration may be performed by policies at or below the object level of abstraction, therefore permitting programmers to make dynamic changes of selected attributes of lower-level runtime libraries and to exploit peculiarities of the underlying multiprocessor hardware. KTK also offers efficient mechanisms for the on-line capture of the program or operating system state required for dynamic configuration.

As an example, consider the *attribute* 'InvocationType', which is a name-value pair:

InvocationType: *enum* {synchr, asynchr}

This attribute expresses that an invocation can be of type 'synchronous' or 'asynchronous'. The code implementing both types of invocations resides in a *policy* object, which in this case, offers the methods 'synchr' and 'asynchr' corresponding to the two possible attribute values. In general, such a *policy* is a special object class that defines, interprets, and enforces the properties intended to be expressed by attributes.

A policy can be associated with any of the program components 'object instance', 'class', 'state variable', 'method', and even 'object invocation'. This association is performed such that the program component's implementation and specification are not affected. For the attribute 'Invocation-Type' above, the policy class containing code for implementation of synchronous vs. asynchronous invocations may be associated with some object instance or class, and the runtime specification of the 'synchr' vs. 'asynchr' attribute value is interpreted by that policy and therefore, determines the semantics of object invocation.

The mechanisms for configuration in KTK can also be used for kernel customization by specializing and/or modifying existing kernel abstractions. In addition, KTK is *extensible* in that new abstractions and functionality (ie., classes, policies, and attributes) are easily added while potentially maintaining a uniform kernel interface (e.g., when not adding any new kernel classes). In our past work, we have developed a significant extension of KTK addressing real-time applications[9], and we are now developing a second extension addressing configurable communication protocols[15]. However, in this paper, we are focussing on the use of KTK's mechanisms for dynamic program and kernel configuration. The kernel abstraction configured in this paper is a mutex lock. The program abstraction configured in this paper is a global queue. As with other global abstractions in parallel programs (e.g., global sums, etc.[24]), certain dynamic program characteristics (e.g., the natural orderings between queue elements due to their times of insertion into the queue) can be exploited in order to reduce the costs of executing certain policies associated with those abstractions (e.g., the amount of work performed or the actual communication topology used to maintain some acceptable global ordering in a queue).

2.2 KTK abstractions and structure

KTK Abstractions. In KTK, an application program consists of a number of independent objects which interact by invoking each other's operations (methods). Each object maintains its own state and that state is not directly accessible to other objects. Objects can range from *light-weight* procedure-like entities to multi-threaded servers with associated concurrency control and scheduling policies. *Complex* objects can be built by having objects as components of other objects, starting with four built-in object classes chosen due to their usefulness in a wide variety of parallel applications constructed with KTK: `'ADT'`, `'TADT'`, `'Monitor'` and `'Task'`[1]. An `'ADT'` (abstract data type) defines an object that has no execution threads of its own and doesn't synchronize concurrent

[1] The built-in object classes in KTK are quite similar to concurrent object constructs offered in recent designs and implementations of object-oriented concurrent languages.

calls. Calling an `'ADT'` is performed in the address space of the caller. Calling a `'TADT'` (threaded abstract data type) creates a new execution thread for execution of the called operation, but also does not synchronize concurrent calls. A `'Monitor'` [10] is an object without execution threads that only allows a single call to be active at a time. It can also define *condition variables* on which calls can *wait*, thereby allowing other calls to proceed, until the condition variable is *signaled*. A `'Task'` (like Ada tasks) has a single execution thread. It defines a number of *entries* which can be called from other objects. All calls are performed in the context of the `'TASK'` and are taken one at a time.

Typical KTK programs consist of complex objects constructed from the four built-in object classes. KTK can be extended by defining new policy classes and linking them to the kernel. In addition, for configurability and to achieve uniformity of kernel interfaces, two distinct views of each object exist: (1) the *application view* and (2) the *system view*. The *application view* of objects is presented in terms of their classes characterizing their external interfaces (methods), where a *class* is an abstraction for a number of similar objects. The *system view*, on the other hand, is defined by each object's *policy* and *attributes*. Essentially, policies define a parameterized execution environment for objects in terms of *attributes*, *invocation semantics*, and *kernel interactions*:

- A policy interprets attributes defined at the time of creation for classes, objects, states and operations. A sample attribute for a class is one that describes some dynamically determined aspect of the internal representation of each of its instances, such as the 'number-of-spins' performed by a lock object before the caller is blocked.

- A policy can define the invocation semantics to an object by intercepting invocation requests and by defining and interpreting invocation time attributes that can be specified as part of the invocation. A sample invocation attribute is one that specifies dynamically determined limits on the permissible duration of an invocation for multi-media or real-time applications[9].

- A policy can also extend the KTK interface with special services. For example, a policy could specify a new invocation mode, such as an *event_dependent_invoke*, defining it in such a way that the other objects could interact with the policy for determining how (or whether) such invocations should be executed. Such control is useful for protecting programs against excessive number of events during emergency conditions, for example.

The policies associated with a program are the vehicles for interactions of application programs with the lower levels of the multiprocessor kernel or with the user-level runtime library supporting KTK programs. Since policies are executed implicitly as a result of object creation and invocation, typical application programs see only the objects and classes defined in their code and offered by KTK. One exception to this rule is when an object explicitly invokes an operation of its own policy (referred to as a *policy interaction*). Such interactions are useful for program-driven dynamic changes to objects.

KTK Structure. The structure of the Kernel Toolkit is depicted in Figure 1 as consisting of three components: (1) configurable threads, which is the portable cthreads package underlying KTK, (2)

Figure 1: Structure of the Kernel Toolkit (KTK)

Configurable threads is the partially machine dependent component[20] that implements the basic abstractions used by the remainder of KTK: *execution threads*, *virtual memory regions*, *synchronization primitives*, *monitoring support* for capture of parallel program and KTK state, and a limited number of *basic attributes* for the configuration of threads-level abstractions, such as synchronization primitives and low-level scheduling. Configurable threads are explained further in Section 5.

The set of policies constructed with KTK varies with the target application domain or hardware. The most complex set is called CHAOS^{arc} and addresses real-time systems[9]. The most commonly used policies are those that offer varying invocation semantics and object representations.

The various object classes required by an application or application domain reside at the application level. Typically, these are *complex objects*, which means that they have associated policies and multiple component objects. An example of a complex object is an internally parallel object containing multiple TADTs used as servers, with a policy that intercepts all invocations to the object. Another example explained in the next section is a complex object implementing a global queue object internally consisting of multiple ADTs serving as distinct object fragments.

KTK implementation. KTK currently runs as a user-level library on various parallel machines, including a 32-node BBN Butterfly parallel processor, a 32-node Kendall Square supercomputer, SUN Sparcstations, and SGI multiprocessors. Previous prototypes of KTK have been run on a larger number of Unix machines, including a Sequent Symmetry, Sun3's, and Sun386. It has also been run as a native kernel on the GP1000 BBN Butterfly multiprocessor.

3 A configurable distributed queue

In this section, we demonstrate how *policies* and *attributes* can be used to configure the behavior of objects during execution. As an example, we describe a specific complex object – a *distributed queue* object – used in our implementation of a best-first LMSK branch-and-bound algorithm solving the Travelling Salesperson Problem (TSP).

Extensive experimental results with alternative TSP implementations on distributed memory (an Intel iPSC machine[23]) and on shared memory machines (the BBN Butterfly and Kendall Square NUMA multiprocessors[6]) have demonstrated that the dynamic configuration of selected program attributes in the TSP application is essential for good runtime performance. This paper provides two different demonstrations of runtime gains due to on-line configuration. The first demonstration concerns runtime adjustments to the global 'work queue' abstraction maintained in the TSP program, as described in this section. The second demonstration concerns additional performance gains due to the dynamic configuration of the mutex locks used within the 'work queue' abstraction, as described in Section 5.2.

The global queue. The TSP program's branch-and-bound algorithm performs optimization in a dynamically constructed search space. The parallel implementation of the TSP application exhibits two abstractions shared by all of its objects: (1) a global best tour object, which is used for pruning the search space, and (2) a work sharing or 'queue' object used for the dynamic distribution of work among the searcher objects. Runtime configuration addresses the queue object. Specifically, we have shown that adequate performance on large-scale parallel machines can be attained only (1) if the queue is implemented as multiple queue fragments distributed across the different nodes of the multiprocessor[24, 6], and (2) if some desirable global ordering on queue elements is maintained. This implies representing the global queue as multiple fragments using some explicit topology as a fragment communication structure and using some program-dependent access policy to each fragment as well as to neighboring fragments.

Queue communication topology and access policy should be represented with KTK policies and attributes, in part because neither of these characteristics affect the basic functionality of such a queue. As in any object-oriented software description, such functionality is represented by the abstraction's operations: *PutSubProblem* and *GetSubProblem*, where the former adds a partial tour to one distributed queue's fragments and the latter returns a previously enqueued element. To summa-

rize, three KTK attributes may be used to specify queue characteristics related to its performance:

- *Topology* – the topology of the communication structure among queue fragments (e.g., a ring, a tree, etc.); and
- *InsertionType* and *WhichPriority* – which are indications of whether a queue insertion should involve other than the local fragment and of the ordering criteria to be used for element insertion, respectively.

An additional, useful queue configuration attribute is the specification of the method of queue invocation (e.g., synchronous or asynchronous). This has been shown useful in non-shared memory multicomputers[24], where a retrieval from a remote fragment may be sufficiently slow so that the invoking object should first complete some other work before checking for the arrival of the sought queue element.

The following Braid[2] pseudo-code demonstrates the specification of policies and attributes using the KTK:

```
POLICY DistributedQueue IS
  ATTRIBUTES
    Topology: TopologyDescription;
    InvocationType: enum {synchr, asynchr};
    InsertionType: enum {local_fragment,
            with_load_balance};
    WhichPriority: PrioritySpecification;
  END
  OPERATION invoke
    ...
  OPERATION load_balance
    ...
BEGIN
  /*initialization code for policy obj creation*/
END
 CLASS Prty_queue_fragment IS
  /* data structure definition for item_type */
  STATE
    /* definition for priority queue */
  END
  OPERATION insert_item ...
    /* insertion code for local fragment */
  OPERATION remove_item
```

[2] *Kernel Tool Kit* objects are described using the Braid Language, which extends the C language with object-oriented constructs and features for expressing attributes, invocation semantics, invocation control, and kernel interactions.

```
  ⋮
BEGIN
  /* initialization code for object of this class */
END
 DistributedQueue CLASS Distr_tsp_queue IS
  STATE
    nb_of_fragments: int;
    /* pointer to Prty_queue_fragment */
    array_of_fragments : *object_t;
    /* priority of the first queue (frag zero) */
    first_priority : int;
    /* index of the first fragment with items */
    first_not_empty : int;
    /* mutual exclusion on the shared queue */
    mutex_t queue_lock;
    ⋮
  END
  CONDITION DoneOrQueueNotEmpty;
  OPERATION GetSubProblem
            (node : OUT object_t);
  OPERATION PutSubProblem
            (node : IN object_t);
  OPERATION init ();
BEGIN
/* initialization code ... */
END
```

In this example, *DistributedQueue* is the policy associated with the class 'Distr_tsp_queue'. When an object q of class 'Distr_tsp_queue' is created, a policy object using the *DistributedQueue* specification is also created and will act as an interface of q to other KTK policies and KTK internal code. Object q's attributes are stored and manipulated by the policy object, and they will be used to dynamically control the execution of q's operations (*GetSubproblem*, *PutSubProblem*, and *Init*).

The state definition for class 'Distr_tsp_queue' describes an object q of this class as composed of a number of fragments ('Prty_queue_fragment's objects), and it also describes general information like the number of elements in the queue (*nb_item*), monitoring information (*insert_time*, *remove_time*), etc. The CONDITION declaration associates with each object the *condition Done-*

OrQueueNotEmpty on which *worker* objects can *wait* or *signal*.

The *DistributedQueue* policy determines the list of attributes associated with q. In this example, four such attributes are enumerated. The class initialization code specifies initial values for these attributes.

Different possible invocation types for objects using the policy *DistributedQueue* are specified in the policy using the OPERATION 'invoke'. This operation's code will examine the value of the attribute 'Invocation_Type' and call either of its internal functions 'invoke_async' or 'invoke_sync' before invoking q's operations *GetSubProblem* or *PutSubProblem*. Specifically, the operation:

INVOKE q\$PutSubproblem (n) <InsertionType=with_load_balance, WhichPriority=tour_value>

will be intercepted by the policy object, its attributes will be evaluated, and then the insertion will be performed as determined by the attributes 'InsertionType' and 'WhichPriority' using the default topology of the policy for communication among the queue fragments.

4 Implementation and performance of configurable objects

Policies. Attributes and policies are the Kernel Toolkit's configuration mechanisms. KTK policies can potentially execute in kernel space, and they have access to low-level library and hardware resources. Any policy written with KTK must interface both with the object it manages and other objects. Toward this end, each policy must offer a number of operations using a fixed naming convention. These operations address:

- *Object creation:* a policy must handle object creation requests, which involves setting object creation time attributes and the actual creation of the user object. This is achieved by invoking relevant operations in the policy object such as *set_state_attributes*, *set_objects_attributes*, and *set_operation_attributes*. Next, the *create* operation for the policy is executed, finally invoking the basic operation *object_create*.

- *Invocation interpretation:* a policy intercepts and performs invocation requests, thereby defining and enforcing invocation attributes and semantics. More specifically, an invocation request of the form:

  INVOKE q\$*PutSubProblem* (n) <Topology =t> is mapped to a call to an operation *invoke* of the policy *DistributedQueue*. The arguments to this operation are contained in an *invocation block* comprised of generic information such as object name (q), operation name (*PutSubProblem*), and a pointer to the actual parameter block containing the argument n. The rest of the arguments to the operation are the invocation *attributes*. The support for various invocation semantics with different sets of attributes is implemented by mapping invocation requests of the form:

  INVOKE\$*mode* obj\$op (*args*) *attributes*

  to the policy invocation:

INVOKE policy(obj)$invoke_*mode*

(invocation_block(obj,op,*args*), *attributes*)

Basic costs. The dynamic use and interpretation of attributes by policies results in certain overheads experienced with object invocations. The measurements depicted in Tables 1 and 2 list the actual overheads experienced on a 32-node GP1000 BBN Butterfly and a 64-node Kendall Square supercomputes, respectively. For reference, a procedure call without parameters to locally stored code costs approximately 3 microseconds on the BBN Butterfly, and a call to a local ADT costs only 18 microseconds. All measurements below distinguish 'local' from 'global' since non-replicated addressing information and attribute values may be located on either the processor's local memory or in remote memory units.

When inspecting Table 1 below, it is apparent that policies impose only small additional overheads on invocations. The reasons for this are the following. First, it takes very little time to locate an object's policy (find policy) since KTK does not attempt to dynamically link new policies to objects after their creation. Similarly, policy invocation itself is cheap since 'small' policies are efficiently constructed with ADTs, therefore not involving additional context switches, etc. In fact, policy invocation cost is dominated by the number of attributes passed to a policy. This is shown by the two different sets of measurements for building invocation blocks involving no parameters (labelled 'fast') or any number of parameters (labelled 'asynch'). In the 'fast' case, none of the mechanism used for space reservation for actual parameters is used during parameter block construction, and no attributes are passed. The 'asynch' case uses those mechanisms (in these particular measurements not involving any actual parameters), but also does not pass any attributes. The measurements for the KSR machine (Table 2) demonstrate this fact further by measuring the cost of 'asynch' invoke with different numbers of attributes, as required for the distributed queue in the TSP application. Of course, none of the measurements below take into account the actual costs of policy execution, which depend on the code contained in policies.

operation	asynch		fast	
	local	global	local	global
find policy	10	30	10	30
build invoc block	40	60	15	25
invokr policy	26	55	26	55
total	76	145	51	110

Table 1: Timing (μseconds) for invocations on GP1000

DSA Objects. The measurements shown above demonstrate that the basic overheads of attributes and policies are not high. The issue addressed next is whether those overheads are outweighed

operation	no attr		1 attr		2 attr		fast	
	loc	gl	loc	gl	loc	gl	loc	gl
fnd plcy	3	12	3	12	3	12	2	12
bld inv bl	3	13	3	14	5	15	3	12
invk plcy	8	30	8	31	9	34	8	30
total	14	55	14	57	17	61	13	54

Table 2: Timing (μseconds) for invocations on KSR1

by the benefits of on-line configuration. For the TSP application, we have evaluated prototype implementations of the work sharing queue on the BBN and KSR multiprocessors, contrasting four different implementations:

1. *Centralized* – the work sharing queue is implemented without KTK as a totally ordered queue located in shared memory and accessed by all worker threads.
2. *Distributed* – the queue is implemented in shared memory as a set of fragments that do not maintain any global ordering and do not perform load balancing.
3. $Distributed_Q$ – this shared memory implementation offers load balancing and enforces a desirable global ordering on queue elements, using a ring-structured communication topology connecting all fragments[6] [3] , again using shared memory.
4. $Distributed_{DSA}$ – a partial prototype of KTK is used to implement $Distributed_Q$, in order to evaluate the cumulative performance effects of the KTK mechanisms of attributes and policies on the TSP application. In the final paper, actual measurements with a full KTK implementation will be presented.

Table 3 lists the execution times of the TSP application (with 15 processors on the BBN Butterfly and 15 processors on the KSR1 multiprocessor) using the four alternative implementations of the work sharing queue explained above. It is apparent from those results that the alternative heuristics (e.g., no load balancing vs. load balancing enforcing a partial global ordering) for queue access and management enforceable with KTK attributes and policies strongly affect program performance. In fact, use of the attributes 'InsertionType' and 'WhichPriority' and the load balancing heuristic using them is shown essential for achieving high performance in the TSP application. Speedup results (on a BBN Butterfly multiprocessor[6]) comparing a non-load balanced with a load-balanced queue shown in Figure 2 further support these statements. At the same time, KTK overheads cause some degradation in performance, as seen by the *DistributedDSA* versus *DistributedQueue* measurements in both Table 3 and Figure 2. Actual execution times on the KSR1 are comparatively smaller to those on the BBN Butterfly due to the KSR's faster processors. KSR results are attained with an untuned prototype version of KTK.

[3] Every two 'get' operations by a searcher thread on its local queue fragment trigger a move of the second best node from the local queue fragment to the next fragment along a ring-structured communication topology connecting all fragments. As a result, 'good' nodes are frequently shared among different searcher threads. This increases the overall quality of nodes used by searcher threads, but it also increases the total number of accesses made by threads to non-local node representations.

Queue Implementation	BBN	KSR1
Centralized	3445	724
Distributed	6920	1346
$Distributed_Q$	2393	579
$Distributed_{DSA}$	2725	636

Table 3: TSP Execution Time (millisecs.) of alternative queue implementations on 15 processors

We posit that similar results will hold for many other parallel application programs. In addition, while some performance degradation occurs when the KTK prototype implementation is used, we believe that the increased ease of programming by use of KTK outweighs the penalties imposed by these costs.

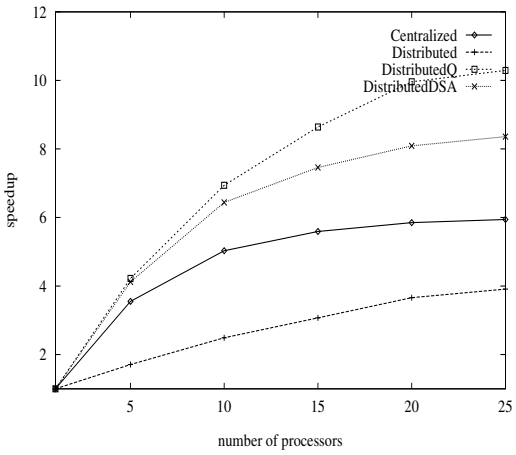

Figure 2: Speedups of alternative implementations of the TSP application

5 Configurable threads

5.1 Performing on-line threads configuration

As with the object layer of KTK, the threads level of KTK also provides support for configurability. Specifically, each component of the configurable threads package defines a set of *basic attributes* used for its configuration. At this level, we are primarily concerned with attributes that characterize a component's internal representation. Such attributes may be altered dynamically by policies resident inside or outside the component. Furthermore, each component also provides support for efficient

customized state monitoring.

Information on component state may be used by heuristics resident inside the component or outside the component (as KTK policies) when making configuration decisions. For example, the configurable lock "queue_lock" used in class "Distr_tsp_queue" in Section 3 contains two sets of attributes and a customized monitor module:

1. The *Wait* attribute set specifies the manner in which a thread is delayed while attempting to acquire the lock. Sample wait attributes are: timeout and spin-time parameters, list of wait methods, etc. These attribute values implement a spectrum of multiprocessor locks ranging from a pure spin lock to a pure blocking lock[21].

2. The *Scheduling* attribute set determines the delay in lock acquisition experienced by a thread using the lock. This set consists of three attributes: (a) a *registration* attribute logging all threads desiring lock access, (b) an *acquisition* attribute 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* attribute that grants new threads access to the lock upon its release.

3. *Custom monitoring* records lock-specific information (*e.g.,* the number of waiting threads at any particular time).

An application uses the customized monitor to sense the current state of the lock, and configures the lock implementation by altering the lock attributes.

5.2 Performance results of on-line lock configuration

Extensive measurements of on-line lock configuration with the TSP and other parallel application programs are reported in [21, 22]. Here, we simply reproduce one set of measurements (Table 4) showing total performance improvements of approx. 20% on the BBN Butterfly and 10% on the KSR multiprocessor when the central queue implementation of TSP uses a configurable rather than a simple spin or blocking lock. Essentially, dynamic lock configuration 'selects' the appropriate lock implementation during execution of the TSP program.

Analysis of locking patterns of the parallel programs being used demonstrates the substantial changes in locking behavior experienced by these applications over time. On-line lock configuration exploits these changes, thereby continually correcting for potential application program and operating system mismatches[21, 22]. Two such locking patterns for the TSP queue's mutex lock are shown in Figures 3 and 4, depicting both heavy contention on the lock and substantial changes in lock contention over time. Extensive studies of locking patterns of multiprocessor applications[22] have shown that: (1) the locking pattern of a multiprocessor lock varies over the lifetime of the application, (2) locking patterns of different locks in the same application are different, and (3) the locking pattern of a particular lock belonging to a particular application differs when run on different machines (as shown in Figures 3 and 4 for 'qlock' on the BBN versus KSR machines). These

observations prompt us to believe that a rigid set of non-configurable synchronization mechanisms will not be able to optimize program performance for different applications and across different machines.

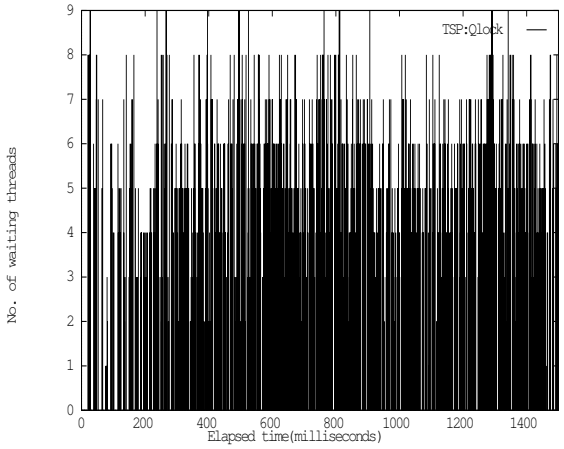

Figure 3: Locking pattern for qlock in TSP; 10-processor run on KSR1

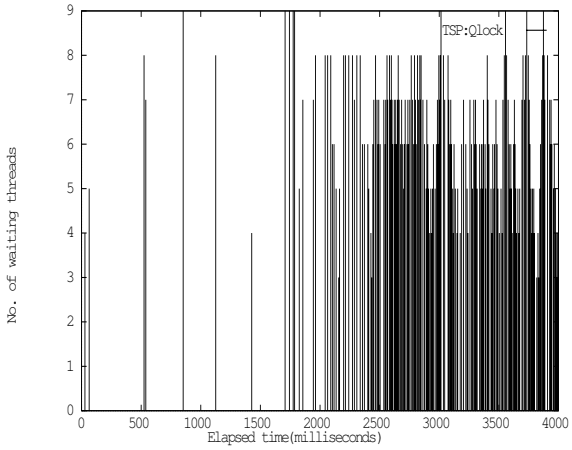

Figure 4: Locking pattern for qlock in TSP; 10-processor run on BBN

The last issue addressed in this paper are the limitations of on-line configuration as per the costs of the state monitoring required for making configuration decisions. Toward this end, we present an evaluation of lock configuration performed with the custom monitoring used in the measurements above vs. an outside configuration policy using explicit object state monitoring in order to configure

Multiprocessor	blocking locks	configurable locks
BBN	3207	2636
KSR1	364	335

Table 4: Performance (millisecs) of the centralized implementations of TSP using reconfigurable locks (using 10 processors)

the mutex lock.

Thread monitoring. Application monitoring and visualization requires more information about a program's execution than can be provided by automatic tracing facilities. Collection of this data generally involves software level instrumentation which may be placed in the operating system, the run time system, system supplied libraries, libraries used as alternatives to system-supplied libraries, or, most often, in the source code of the program under study. The "thread monitoring" system developed by our group provides general monitoring facilities at the threads level of KTK.

Monitoring at the thread level is implemented using a dedicated "local monitor" thread that collects trace data (sensors) from application threads, performs some low-level processing if necessary and sends traces to a "central monitor" if desired (possibly running in a remote machine). The central monitor uses an X window-based "user interface" for users to communicate with the monitoring system.

Application threads communicate with the local monitor thread using thread-specific monitor buffers. In an un-optimized implementation of the system on the KSR1, creation of a 40-byte long lock-sensor (containing 'lock type', 'time-stamp', 'current processor number', 'thread id', and 'lock state') takes 118 μseconds, insertion of the sensor in the thread's monitor buffer takes 58 μseconds, and retrieval of the sensor by the local monitor thread takes 51 μseconds. These measurements do not include costs for cache line movement across processors.

We found the performance of such a general monitoring scheme unsuitable for implementation of the configurable lock (for comparison, a mutex lock takes 6 μseconds on the KSR1), in part because of the loose coupling between the application thread and the local monitor thread (it takes 38 μseconds for communication of a 8 byte word from an application thread to the local monitor thread). This causes performance degradation and configuration delay[4]. Hence, for the configurable locks described above, we use a closely coupled, customized monitor which monitors only those lock-specific attributes that are required for on-line lock configuration (*e.g.,* no. of waiting threads, lock holding time, etc). Furthermore, we eliminate the local monitor thread and employ the application thread calling the lock for its monitoring and configuration. With this scheme, it takes 5.5 μseconds to monitor and configure an attribute for a configurable lock. This implies that KTK must offer a variety of monitoring and configuration mechanisms for use with different configuration tasks. Such

[4] A configuration action is based on lock state that is 'too old' due to the relatively slow and asynchronous communication between applications threads and the local monitor thread.

variety is offered in KTK by permitting alternative implementations of policies (such as ADTs, TADTs, etc).

6 Conclusions

KTK and configurable threads provide an efficient basis for building configurable operating system kernels and application programs for multiprocessor systems. The performance advantages demonstrated with on-line configuration range from 10%-50%, for a variety of parallel application programs and abstractions, on a BBN Butterfly and a KSR multiprocessor. For larger-scale parallel application programs, we are hoping to achieve cumulative performance gains approaching or exceeding 100%.

The general goal of our group is extend the notions and mechanisms of configuration presented in this paper to develop a new technology for high performance computing systems, called *interactive program steering*. The essential idea of this technology is to give users the ability to *steer* their programs quickly past uninteresting results or data domains, therefore significantly reducing program execution time or alternatively, offering additional computing power for required high-fidelity computations. In general, such *interactive program steering* can be defined as *the configuration of a program by algorithms or human users during its execution*. We wish to understand the basic principles and opportunities of program steering, to develop abstractions and tools that facilitate the construction, execution, and control of steerable and configurable programs, and to demonstrate the performance advantages of program steering on parallel and distributed target machines.

Evidence of the utility of interactive program steering already exists in many large-scale parallel programs. For example, in our collaboration with physicists, we estimate that the sizes of 30%-40% of the time steps taken by their physical simulations of advanced material properties may be reduced in size and therefore, in computational duration, perhaps in favor of more precision and fidelity of results during interesting material behaviors. Additional reductions in computation time can be achieved by use of the on-line configuration algorithms presented in this paper, which recognize certain program characteristics and reconfigure selected program abstractions and/or operating system mechanisms and policies.

Future work on the topics presented in this paper concerns (1) the evaluation of cumulative performance improvements for applications like the TSP program and also for a large-scale parallel simulation kernel (a Time Warp simulator) and (2) the integrated use of the threads-level monitoring facilities with the KTK library. In addition, the native implementation of KTK on next generation multiprocessor hardware is being considered. Other research conducted by our group addresses the use of KTK for protocol construction as well as the extension of KTK across non-shared memory systems.

Acknowledgements

We would like to thank Nirupama Mallavarupu for implementing the general thread monitoring system, and helping us take measurements of the system on KSR1 multiprocessor.

References

[1] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The eden system: A technical review. *IEEE Trans. Softw. Eng.*, SE-11(1):43–58, Jan. 1985.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

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

[4] Kenneth P. Birman and et.al. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, pages 502–508, June 1985.

[5] Roy Campbell, Vincent Russo, and Gary Johnson. Choices (class hierarchical open interface for custom embedded sstems. *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[6] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (dsa) on large-scale multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-93-25, May 1993. To appear in 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS-IV), Sept. 1993.

[7] George Cox, William M. Corwin, Konrad K. Lai, and Fred J. Pollack. A unified model and implementation for interprocess communication in a multiprocessor environment. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, pages 44–53, Dec. 1981.

[8] Geoffrey C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors.* Prentice-Hall, 1988.

[9] 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.

[10] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[11] 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.

[12] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):77–107, Feb. 1988.

[13] Jeff Kramer and Jeff MaGee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[14] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of the 5th ACM Symposium on Operating System Principles*, Nov. 1975.

[15] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. Technical Report GIT-CC-93/22, College of Computing, Georgia Institute of Technology, March 1993. To appear in the International Conference on Network Protocols,1993.

[16] J. MaGee and J. Kramer. Dynamic configuration for distributed real-time systems. In *Proceedings of the International Conference on Parallel Processing*, pages 277–288, Aug. 1983.

[17] Keith Marzullo and Mark Wood. Making real-time systems reactive. *ACM Operating Systems Review*, 25(1), Jan. 1991.

[18] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, Dec. 1989.

[19] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. O/s support for portable lightweight threads. Technical Report GIT-CC-93-53, College of Computing, Georgia Institute of Technology, August 93.

[20] Bodhisattwa Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *Proc. of the International Conference on Parallel Processing*, volume 2, pages 205–208, Aug. 1993.

[21] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 59–66, July 1993.

[22] 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*, pages 191–218, Dec. 1989.

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

[24] Karsten Schwan and Anita K. Jones. Flexible software development for multiple computer systems. *IEEE Transactions on Software Engineering*, SE-12(3):385–401, March 1986.

[25] 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.

[26] J. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 146–157, Dec. 1987.

[27] A. Tucker and A. Gupta. Process control and scheduling issues on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159–166, Dec. 1989.

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