CTK: Configurable Object Abstractions for Multiprocessors

Dilma M. Silva

Computer Science Department
University of São Paulo
Rua do Matão, 1010
CEP 05508-900 São Paulo, Brazil
dilma@ime.usp.br

Karsten Schwan
Greg Eisenhauer

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{schwan,eisen}@cc.gatech.edu

Abstract

The *Configuration Toolkit* (CTK) is a library for constructing configurable object-based abstractions that are part of multiprocessor programs or operating systems. The library is unique in its exploration of runtime configuration for attaining performance improvements: (1) its programming model facilitates the expression and implementation of program configuration, and (2) its efficient runtime support enables performance improvements by configuration of program components during their execution. Program configuration is attained without compromising the encapsulation or the reuse of software abstractions.

CTK programs are configured using *attributes* associated with object classes, object instances, state variables, operations, and object invocations. At runtime, such attributes are interpreted by *policy* classes, which may be varied separately from the abstractions with which they are associated. Using policies and attributes, an object's runtime behavior may be varied by (1) changing its performance or reliability while preserving the implementation of its functional behavior or (2) changing the implementation of its internal computational strategy.

CTK's multiprocessor implementation is layered on a Cthreads-compatible programming library, which results in its portability to a wide variety of uni- and multi-processor machines, including a Kendall Square KSR-2 Supercomputer, SGI machines, various SUN workstations, and as a native kernel on the GP1000 BBN Butterfly multiprocessor. The platforms evaluated in this paper are the KSR and SGI machines.

Index Terms: configurable systems, high performance objects, object fragmentation, dynamic adaptation, complex parallel programming, library operating systems

This research was supported in part by CNPq-Brazil (under contract 352737/96-4) and DARPA (under contract DABT63-95-C-0125), and NSF (with Research Infrastructure grant #9422033).

1 Attaining High Performance by Dynamic Configuration

1.1 Background

To attain high performance, scalability, and extensibility, two important properties of application programs and operating system components are (1) the *separation* of basic component functionality from aspects governing performance and (2) the *configuration* of those aspects to maintain high performance despite variations in a component's use and execution environment. For example, for complex multi-media and real-time applications, Quality of Service (QoS) requirements and the resource allocation functions used to enforce these requirements are routinely separated from the applications' basic functionalities. Furthermore, runtime adjustments to requirements, service levels, and to associated resource allocations are frequently necessary. For distributed media applications, such functionality is exemplified by enhancements of communication mechanisms with quality of service parameters and with policy-level (policing and bandwidth reservation) functionality[18, 19]. For real-time and embedded applications, researchers have developed languages for stating service contracts and methods for adapting programs at runtime to meet such contract obligations[42, 43]. In addition, there are formulations and implementations of objects and invocations that are able to meet timing attributes [46] or more generally, QoS objects [57] that permit the association of arbitrary service contracts with objects and object invocations. Similarly, research in high performance computing and in operating systems has derived gains in performance and reliability from the dynamic configuration of multiprocessor synchronization constructs[9, 36], of network communications[27, 26], of file systems[39], and of distributed object abstractions[12, 3].

Separation and configuration can also be the means of maintaining backward compatibility with existing applications' type-dependent interfaces or with previous versions of operating systems, neither of which should change when developers offer additional functionality. This is exemplified by the replication of file objects in the Spring operating system[24], by a location-transparent object invocation mechanism in the Lipto system[15], and by object migration on uniprocessor platforms in Apertos[56].

Performance improvements derived from program configuration typically correct mismatches between desired vs. current program or system primitives, policies, and state. Since such mismatches may arise at any time during a program's execution, configuration mechanisms should be designed to support both off-line and online configuration. For example, since system loads cannot be predicted in multi-user systems, the optimal levels of parallelism (i.e., scheduling) for a parallel program can only be determined and set dynamically[51]. Similarly, runtime changes in lock implementations on multiprocessor programs can result in substantial performance gains[36]. This is because the size of the critical section protected by the lock and the resulting contention in critical section access are subject to frequent and unpredictable change during program execution.

1.2 Contributions

The object-based programming library, called *Configuration Toolkit* (CTK), described and evaluated in this paper demonstrates (1) the separation of objects' functionalities from their performance or reliability attributes and (2) the configuration of objects in response to changes in those attributes:

- CTK permits the specification and runtime use of (configuration) *attributes* for object classes, object instances, state variables, operations, and object invocations.
- Attributes are interpreted by system- or programmer-defined *policy objects*, which may be varied separately (at link-time) from the abstractions with which they are associated. A policy may interpret runtime-supplied attribute values to vary an object's internal implementation without changing its functionality, or it may vary the semantics and implementations of object invocations without affecting or altering the methods being invoked.
- Dynamic configuration may be performed by policies at or below the object level of abstraction, thereby also permitting programmers to change selected attributes of lower-level libraries at runtime (sometimes called 'translucence' in recent research[57]) and to exploit peculiarities of the underlying multiprocessor hardware.

CTK also provides efficient mechanisms for the on-line capture of the program or operating system state required for dynamic program configuration.

The experimental results attained with CTK on multiprocessor platforms and presented in this paper demonstrate the viability and importance of using object-based techniques to construct configurable parallel programs, where the purpose of configuration is to attain suitable levels of performance on parallel machines. In the remainder of this paper, we first describe the fashion in which basic object functionality may be separated from its configurable attributes, by designing a sample configurable multiprocessor object. The 'distributed queue' object described in Section 3.1 supports the queue access policies and the queue implementations that result in high performance for the programs using it. Object configurations are specified with simple language constructs, and their implementation is facilitated by certain pre-defined classes and policies offered by CTK, thereby obviating the need for end users to know the internal implementation of CTK's runtime support. Second, using the distributed queue example, specific configuration behaviors implemented with policies and attributes are shown to result in significant performance gains on various target multiprocessor platforms. Such gains are enabled by the low overheads of CTK's runtime support measured in Section 4.2; they are attained by simultaneous use of multiple configuration techniques, including (1) the dynamic interposition of configuration code between object invoker and implementation, (2) the parameterization of selected object characteristics, and (3) object fragmentation and the efficient runtime maintenance of such fragments. Third, policies and attributes are used to *extend* object functionality to utilize typical kernel-level operating system abstractions and services. Sample extensions described in Section 4.3 'convert' non-persistent to persistent objects transparently to end users, and they configure objects such that all of their invocations are monitored automatically.

2 Related Research

2.1 Operating Systems and High Performance Computing

Both the operating system and the high performance computing communities have used program configuration to improve performance and/or reliability. Historically, high performance applications have dealt with potential static or dynamic mismatches with the operating system by entirely removing it from the runtime environment, as evident in early hypercube machines that only offered operating system support for process creation and message passing. A similar reaction has been to minimize the runtime use of operating system facilities to those that offer suitable performance, often sacrificing ease of programming[55], or to tailor programs to underlying hardware and systems using compiler support[21].

In comparison to past work, we aim to achieve improved programmability, extensibility, and scalability of performance-sensitive programs by use of runtime configuration. For operating systems, such configuration has been made possible by a large body of past and current work, including early work on policy-mechanism separation[13], the removal of operating system services from 'fixed' kernels to the configurable user level in systems like Mach[40] and Windows NT, and most recently, the development of micro-kernels, library operating systems[10, 17], and dynamic linking and binary editing techniques[33]. Such research has established the fact that application programs should be 'combined' with operating system functions such that both may be configured jointly, using the same programming techniques and software infrastructures.

The joint configuration of application- and system-level functionality has also been explored in several recent object-based efforts, including the Choices[7], Spring[24], Chaos[22], and Apertos[56] operating systems, and it has been investigated in efforts that address object fragmentation [12, 25], and fragmented network objects[48].

Program or operating system configuration may be performed with diverse methods and at different times, including at compile-, link-, boot-time[10, 17, 7]. Runtime configuration methods include the dynamic re-linking of OS kernels, server process creation and deletion[40], the re-direction of selected system calls using interposition[28] and subcontracts [24], the dynamic adjustment of meta-objects[44], the dynamic adaptation of object and invocation implementations[37], and the runtime adjustment of structural properties of sets of communicating objects[1]. Similarly, recent work on the dynamic configuration of distributed and object-based systems[8] concerns specific configuration methods or general (rather than high performance) frameworks for implementing dynamically configurable applications. Compiler-based techniques for runtime configuration have included the dynamic synthesis of program code and the re-direction of procedure calls[39].

Our aim is to enable programmers to use any number of configuration techniques and to use those techniques to improve the performance of parallel programs. Our approach is to give programmers a flexible library with which they may construct dynamically configurable and efficient software abstractions for a variety of target parallel machines. Toward this end, we have developed the CTK library for construction of configurable objects, which runs on a variety of parallel machines due to its layering on top of a portable parallel programming library, configurable Cthreads[36]. Cthreads programs have run on BBN, KSR, Sequent Symmetry, and Encore Multimax multiprocessors and are currently running on SGI, SUN Sparc, and x86-based parallel machines. This paper reports on results that utilize the KSR and SGI machines.

2.2 Real-time and Object-based Systems

CTK's emphasis on the attainment of improved performance in parallel systems distinguishes our work from past efforts addressing distributed or real-time systems[5, 29, 34], where the primary concern has been to maintain certain levels of system responsiveness or reliability in the presence of uncertain execution environments. It also distinguishes this research from our own past work[4, 22], in which we investigated how the needs of real-time applications affect the object model and its required runtime support . Last, in comparison to ongoing work addressing the online adaptation of distributed multimedia or embedded applications[42, 43, 45], such work is typically emphasizing the development of adaptive methods for certain classes of applications. Such methods may be realized in CTK using its notions of attributes and policies.

Recent work on reflective programming[20, 11] and on Quality of Service objects[57] is now leading researchers to offer abstractions like 'meta objects' or 'delegates', which are similar to the policy objects used in CTK. Both approaches provide mechanisms for changing an object's behaviors dynamically in most of its aspects (object creation, method invocation, etc), but in general, meta-object protocols and delegates address configuration problems for which efficient runtime configuration is not crucial. For example, in Apertos[56], meta-objects are used to achieve object heterogeneity such that changes in object behavior persist despite *object migration* among different meta-object spaces. The required runtime checking of meta-object compatibility is too computationally expensive for applications in which configuration may be highly dynamic or short-lived. Similarly, in BBN's Quo system[57], the current realization of delegates relies on the use of Java technologies in conjunction with the CORBA-compliant object systems being configured. Such realizations are not suitable for the target parallel machines and programs addressed in this paper. In the development of ABCL/R3, Yonezawa, Masuhara and Matsuoka [32, 31] were concerned with highly parallel computing, but their focus was on providing specific support for customizable resource management policies (for example, by providing the notion of nodes explicitly at the meta-level), while our work aims also at object customizations regarding algorithmic strategies. Besides, in ABCL/R3 the customization is achieved through a chain of operational definitions of how to interpret base-level expressions (*executors*). In general, such operational style of extension is not efficient. In SOM[14] or CLOS[38], reflection principles are used for changing object behavior, by invoking the methods available to create and initialize classes, to compose their methods tables, etc. Namely, object descriptions may be altered at runtime by invoking the inherited methods that deal with class objects, thereby attaining configuration by manipulation of a potentially complex class description. In comparison, object configuration in CTK is relatively 'lightweight', since it involves only the execution of policy objects that have been specifically created for configuration purposes. This makes CTK more suitable for attaining performance gains via configuration, whereas the reflection principles as realized in SOM and CLOS primarily address Software Engineering issues like the evolution of compatible object libraries by configuration of entire object system structures.

At the heart of CTK is a system for concurrent object programming with features that bear a resemblance to those in systems such as ACT++, CC++, ICC+, Mentat, pSather, ABC++, Concurrent Aggregates and COOL. However, since the focus of this paper is configurability, we limit our review of related research to those systems which address that issue in some way.

3 The Configuration Toolkit (CTK)

3.1 Motivation and Example

A parallel application developed with CTK is described by a set of classes, their policies, and an 'application description' module in which the static objects composing the application are listed. Multiprocessor node locations and initial values for attributes can be specified for these static objects, and additional information needed to generate the executable (libraries, object files, configuration parameters for the underlying portable multiprocessor runtime support) may be provided, as well. Classes are specified using the experimental *Braid*[1] language, which extends the C language with object-oriented constructs and features for expressing attributes, invocation semantics, invocation control, and kernel interactions. 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.

The novel abstractions offered by the *Configuration Toolkit* (CTK) are its *attributes* and *policies*, where attributes may be associated with object classes, object instances, state variables, operations and object invocations, and where policies interpret those attributes to effect runtime configuration. For instance the *attribute* 'InvocationType' stated as a name-value pair:

InvocationType: *enum* {synchr, asynchr}

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.

The association of attributes with any of the program components 'class', 'object instance', 'state variable', 'method', and 'object invocation' is performed such that the components' implementations and specifications are not affected. For example, for the attribute 'InvocationType' above, the policy class implementing synchronous and asynchronous invocations is associated with each object instance being invoked, and the runtime specification of the 'synchr' vs. 'asynchr' attribute value is interpreted by that policy.

The example of a configurable object used throughout this paper is that of a *distributed queue*. The performance of many parallel programs and operating system services (e.g., schedulers[6]) has been shown sensitive to the implementation of the queue abstractions they use. The particular parallel program studied in this paper is a parallel branch-and-bound code, which uses a priority queue to store its subproblems that remain to be solved. The performance of this program strongly depends on (1) the efficiency of queue access and (2) the order in which subproblems are retrieved from the queue, since this guides how the search space will be traversed. Results from previous work demonstrate that both a centralized implementation and one that maintains non-cooperating queue fragments local to each processor lead to poor performance. In contrast, experimentation with alternative implementations of the Traveling Salesperson Problem (TSP) on distributed memory (an Intel iPSC machine[47]) and on shared memory machines (the BBN Butterfly and Kendall Square NUMA multiprocessors[12], and bus-based multiprocessors) have demonstrated that the dynamic configuration of selected attributes of the distributed queue is essential for good runtime performance. In a nutshell, adequate performance on large-scale parallel machines can be attained (1) if the queue is implemented as multiple cooperating fragments distributed across the different nodes of the multiprocessor, (2) if some desirable global ordering on queue elements is maintained across all fragments, and (3) if the access policies to queue fragments and the interconnections among queue fragments can be customized to specific application runs.

[1] Braid is the name of the language used to specify CTK objects. It has no relationship with West and Grimshaw's Braid[53].

We next present some of the basic abstractions offered by CTK, following which we show that it is straightforward to implement an internally fragmented and configurable distributed queue abstraction with CTK's *policies* and *attributes*. Experimental evaluations of alternative global element orderings and of customized queue access policies appear in Section 4.2.

3.2 CTK Abstractions and Their Use

CTK abstractions. A CTK application program consists of a number of independent objects that 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' implementations range from *light-weight* procedure-like entities to multi-threaded servers with associated concurrency control and scheduling policies. *Complex* objects may have other objects as components, starting with four built-in object classes chosen due to their usefulness for a wide variety of parallel applications constructed with CTK: `'ADT'`, `'TADT'`, `'Monitor'` and `'Task'`. An `'ADT'` (abstract data type) defines an object that has no execution threads of its own and doesn't synchronize among concurrent invocations. Invoking an ADT causes the execution of its method in the address space of the invoker. In comparison, invoking a `'TADT'` (threaded abstract data type) creates a new thread for execution of the called operation. A `'Monitor'` 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* that can be called from other objects. All calls are performed in the context of the `'Task'` and are executed one at a time.

The basic functionality of CTK explained in the previous paragraph may be extended by defining new policy classes and linking them with the CTK library. The extensions in the version of CTK employed in this paper include policies implementing multiple invocation semantics, such as asynchronous vs. synchronous invocations and support for policy replication (see Section 3.3), as well as runtime feedback concerning invocation state (i.e., invocation monitoring) and persistence support (see Section 4.3). The implementation of such extensions is straightforward, in part due to their support by CTK's policy and attribute mechanisms and by lower-level library support that shields implementors from the requirement to know intricate details of CTK's implementation when such extensions are made. This suggests that it will also be reasonable for programmers to extend CTK with object representations and invocation semantics that are tailored to certain target architectures and/or to their applications' specific requirements in functionality and performance.

Typical CTK programs consist of complex objects constructed from the four built-in object classes and from the application- or machine-specific CTK extensions. Most relevant to this paper is the fact that CTK offers two distinct views of each object, thereby separating an object's basic functionality from its configuration aspects: (1) the *application view* and (2) the *configuration view*, the latter being similar to recent notions of meta-objects implemented in systems like ACE/TAO[45]. The *application view* of an object is presented in terms of its class, characterizing its external interface (methods). The *configuration view* is defined by the object's *policies* and *attributes*. Essentially, policies define a parameterized execution environment for the object in terms of *attributes*, *invocation semantics*, and *configuration interactions*:

- A policy interprets attributes defined at the time of creation for classes, objects, states, and operations.

- 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 policy can also extend an object's interface with special services.

The implementation of the distributed queue using policies and attributes described next illustrates the manner in which configurable CTK objects are constructed.

Queue implementation with CTK. The class *Item* describes the objects to be manipulated by the queue. *Item* objects have associated priorities. A *Fragment* object holds a local fragment of the global priority queue through a linked list of *Item* objects and a lock to protect the data structure from concurrent access. The operations 'insert' and 'remove' are available in the *Fragment* interface. A *Queue* object encapsulates the fragmented global queue abstraction by keeping the number of fragments and a set of *Fragment* objects. Attributes are associated with the object parts involved in its configuration, and a policy associating the desired semantics to attribute values is specified. Each invocation of a *Queue* method will be intercepted by the policy object, which will interpret the appropriate attributes. The aspects of *Queue* behavior chosen to be configurable are:

1. *The global order of elements in the queue* – five possible schemes for element insertion and removal are implemented and evaluated:

   - **weak-order**: no global order is maintained, which implies that the operation 'remove(item)' simply invokes the method 'remove' on the *Fragment* object local to the processor performing the invocation, and then returns the local element with highest priority (which can have a low priority in a global view). If the local fragment is empty, a removal is referred to some remote fragment, thereby implementing a simple load balancing policy. However, insertions are always executed locally. For both operations, a lock is used to synchronize local operations with (relatively rare) removals on an empty fragment referred to a remote node.
   - **give-second-best**: as with "weak-order", insertions and removals are primarily executed locally by the processor invoking them, but some priority load balance is implemented across the fragments of the distributed queue, by periodic exchanges of queue elements. Namely, each fragment will periodically (e.g., at every ten operations) give its locally second best element to one of its neighbors. The frequency of this exchange is an attribute of this configuration option.
   - **get-second-best**: rather than automatically giving away its second best element, with this method, each fragment will periodically request another fragment's second best element.
   - **share-kth**: in this protocol, each processor holding a *Fragment* object has some information about the k^{th} best elements of all other fragments. This information is used to determine whether a removal should return an item from the local fragment, or whether it should issue a remote access in order to get an item with better priority. The choice of k (first, second, third, etc) and the frequency with which such priority information is updated are configurable.

- **total-order**: a removal always returns the item with highest overall priority. The collection of fragments mimics the behavior of a central queue. The queue policy achieves this by either using a single fragment or by coordinating fragments during 'remove' invocations. Our implementation uses multiple fragments, which facilitates the transition from this ordering protocol to the others described above.

2. *Topology* – the topology of the communication structure among queue fragments (e.g., a ring, a tree, complete graph, etc).

3. *Invocation mode* – may be synchronous or asynchronous. This has been shown useful in non-shared memory multicomputers, 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.

4. *Priority* – a queue element's priority may have several sources. For example, when the queue item represents a subproblem in a branch-and-bound application, the size of the subproblem may be combined with application dependent measures of quality. Our implementation uses different weights for each such contribution to an element's priority, which in turn may be configured dynamically. The judicious use of weights provides another way of guiding the branch-and-bound application while it traverses the search space.

When constructed with CTK, the resulting distributed queue object may be described by the Braid-like pseudo-code in Figure 1.

```
DistributedQueuePolicy class Queue is
    state
        nb_of_fragments: int;
        array_of_fragments : Fragment[ ]        < Topology = CompleteGraph>;
    end
    operation insert (item : in Item)           < Order = shared-kth,
                                                  HowOften = 20,
                                                  K = 3,
                                                  InvocationType = sync,
                                                  ItemWeight = 0.5,
                                                  SizeWeight = 0.5 >
    operation remove (item : out Item);
begin
    /* initialization code for creating Queue objects */
end
```

Figure 1: Queue specification

In this figure, the class *Queue* has the policy *DistributedQueuePolicy*, as specified in the first line in Figure 1. The state variable 'array_of_fragments' has an attribute describing the communication structure among fragments. Here, each fragment may communicate with any other fragment, so that "complete_graph" is specified as an initial value for this attribute. This implies that a 'remove' operation executed on an empty local fragment can access all other fragments (through invocation of their methods) in order to obtain an item. The prototype for operation 'insert' specifies initial values for the attributes related to invocation mode, order of elements, and priority.

When an object q is created, a policy object using the *DistributedQueuePolicy* class specification is also created and will act as an interface of q to other CTK policies and to CTK's internal code. Object *q's* attributes may be stored and manipulated by the policy object, and they will be used to dynamically control the execution of *q's* operations. Assuming the policy implementation detailed in Figure 2, the invocation 'invoke q \$ insert(item)' will be intercepted by the policy object, and the 'shared-kth' order protocol will be used with the values $k = 3$ and $HowOften = 20$. As a result, every time the number of insert/remove invocations local to a fragment exceeds the value 20, the information about this fragment's third best element priority will be passed to other fragments (for the "GetSecondBest" the global number of operations performed is counted).

```
tadt class DistributedQueuePolicy is
    state
        Topology: enum {CompleteGraph, Ring, Tree};
        InvocationType: enum {synchr, asynchr};
        ItemWeight: float;
        SizeWeight: float;
        Order : enum {weak, give-second-best, get-second-best, share-kth, total};
        K: int;
        HowOften: int;
        ...
    end
    operation invoke      (pb: ParameterBlock,
                           <list of attributes associated with invocations >);
    operation invoke_synch (pb: ParameterBlock);
    operation invoke_asynch (pb: ParameterBlock);
    operation set_class_attributes (...);
    operation set_state_attributes (...);
    operation set_object_attributes (...);
    operation set_operation_attributes (...);
begin
    /* initialization code for policy object creation*/
end
```

Figure 2: The *DistributedQueuePolicy* class

The policy object is responsible for storing this shared information, triggering the broadcasting of this information at the appropriate point in time, and using the values it has from all fragments to decide whether a fragment local to the processor issuing the invocation should execute it or activate the invocation in a (better) remote fragment. Attribute values specified via class description or object specification (the create statement) can be overridden during execution by a later invocation, such as 'invoke q\$remove(item)<Order=total>'. This invocation will cause the policy object to adjust its internal state in order to follow the total-order protocol until another value is specified via a subsequent invocation. Alternatively, the total order protocol could last for this invocation only.

CTK objects like the distributed queue may be created dynamically, or their creation may be specified at compile-time using the 'application description module'. In either case, initial values may be specified for the attributes of each particular instance of a class, as exemplified for those specified for the queue object appearing below:

q <Topology = ring> : *Queue*; /* static */

object_t q;   create q <Topology = ring> : *Queue*; /* dynamic */

Policy objects. The actual specification of the *DistributedQueuePolicy* in Figure 2 uses the same syntax and structure as the class *Queue* described earlier. However, several operations are specified in addition to the basic 'invoke' operation for this object. This is because any policy written with CTK must interface both with the object it manages and with other objects it may wish to affect (e.g., lower-level objects accessible to it when it executes in kernel space), and because it must manage attribute values. In the current implementation of CTK, such additional operations use fixed naming conventions, and they address object creation and invocation interpretation.

Object creation. A policy must handle object creation requests, which involves setting creation time attributes and the actual creation of the object instance. This is achieved by invoking the relevant operations in the policy object, including *set_state_attributes*, *set_objects_attributes*, and *set_operation_attributes*[2]. In the distributed queue example, the policy itself is a TADT object, therefore if the class specified for the policy has a policy of its own, then its 'set' operations will be executed in a hierarchical fashion. After the policy object is created, a thread is forked to run the initialization code for this new object, and the CTK object creation code for the original object continues (for example, queue fragments may be created).

Invocation interpretation. A policy intercepts and performs invocation requests, thereby defining and enforcing invocation attributes and semantics. For example, an invocation request of the form:

invoke q\$insert $(item)$ $<Order = total>$

is mapped to a call to an operation *invoke* of the policy *DistributedQueuePolicy*. The arguments to this operation are contained in an *invocation block* comprised of generic information regarding the object type, operation, and a pointer to the actual parameter block containing the argument *item*. The other 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*)

In addition to this static way of specifying invocation modes, the dynamic binding of invocations to invocation modes can be achieved with attributes. Toward this end, policy code associates semantics with attribute values by calling one of the internal CTK functions (e.g., 'invoke_async' , 'invoke_sync', etc).

[2] It would be straightforward to have the Braid compiler generate default 'set' operations in case they are not stated in the class description.

3.3 Policy Replication

The design of the *DistributedQueuePolicy* class described in the previous section is guided by the general idea that the policy implements the high-level coordination of work being undertaken by multiple *fragment* objects distributed across processors, thereby dynamically controlling the global behavior of the queue object. Clearly, it is important that tasks related to such runtime configuration consume as few resources as possible, else configuration overheads would overshadow its benefits. In both shared memory and distributed memory multiprocessors (and even more so in networked systems), small configuration overheads can be attained only if the entire configurable CTK object, including its basic object, its policies, and its parts (ie., the fragments) are perfectly distributed. The *DistributedQueuePolicy* depicted in Figure 1 does not result in such complete distribution, as explained next.

Consider the situation depicted in Figure 3. Here, the distributed queue is mapped to three processors. Processors 1 and 2 each host one of the queue fragments; Processor 3 hosts one fragment, the queue abstraction (*i.e.,* the *Queue q* object), and its corresponding policy. The invocation labeled (1a) – 'insert' – is invoked by the application code running on Processor 1; the invocation (2a) – 'remove' – is invoked on Processor 2; and the invocation(3a) — 'insert' - is invoked on Processor 3. Since these invocations may be performed concurrently, they are all redirected to the policy object (arrows (1b), (2b), and (3b), respectively) residing on Processor 3, which evaluates the attributes describing the current queue configuration and decides how each operation should be carried out. In the situation where the attribute 'Order' has the value *weak*, the invocations will simply be referred to the appropriate local queue fragment (arrows (1c), (2c), (3c)). It should be apparent from this example that this design is unsatisfactory, because it results in remote accesses to the policy object to direct an operation to a locally resident fragment!

To enable locality of object access, CTK provides a built-in 'replicated' object type with which both objects and their policies may be distributed, thereby avoiding the potential bottleneck of a single policy instance used with many distributed object fragments. Namely, when an object is specified as 'replicated', a copy of it will be created on each processor, and references to it will be translated into references to the local copies. Since object replication does not imply consistency guarantees for multiple object copies, programmers may implement object and policy-specific consistency protocols. The implementation of consistency protocols in turn are supported by the pre-defined CTK policy for multi-invocations, which is stated in Braid as 'multi_invoke[3]. This policy ensures that any invocation of a policy copy is automatically sent to all other copies of the policy.

Declaration of the queue *q* as a replicated object changes the design of the *DistributedQueuePolicy* into one that uses a complex object with its components distributed across the same processors on which queue fragments are located. As a result, remote invocations of CTK objects are performed only when necessitated by the queue's distributed implementation, not by the implementation of its policy. More precisely, when a *Queue q* is created, its replicated behavior on creation will cause the generation of multiple *local queues* and *local policies*. Each operation on *q* is intercepted by a local policy object, which may be able to carry out the invocation without invoking objects residing on other processors. For example, for insertions using 'weak-order' or 'total-order', all policy actions and the execution of the queue's basic functionality can be carried out locally.

[3]The statement 'multi_invoke <*array_of_objects*> # *nb_elements* $ *op*' results in invoking *op* in each of the elements in *array_of_objects*.

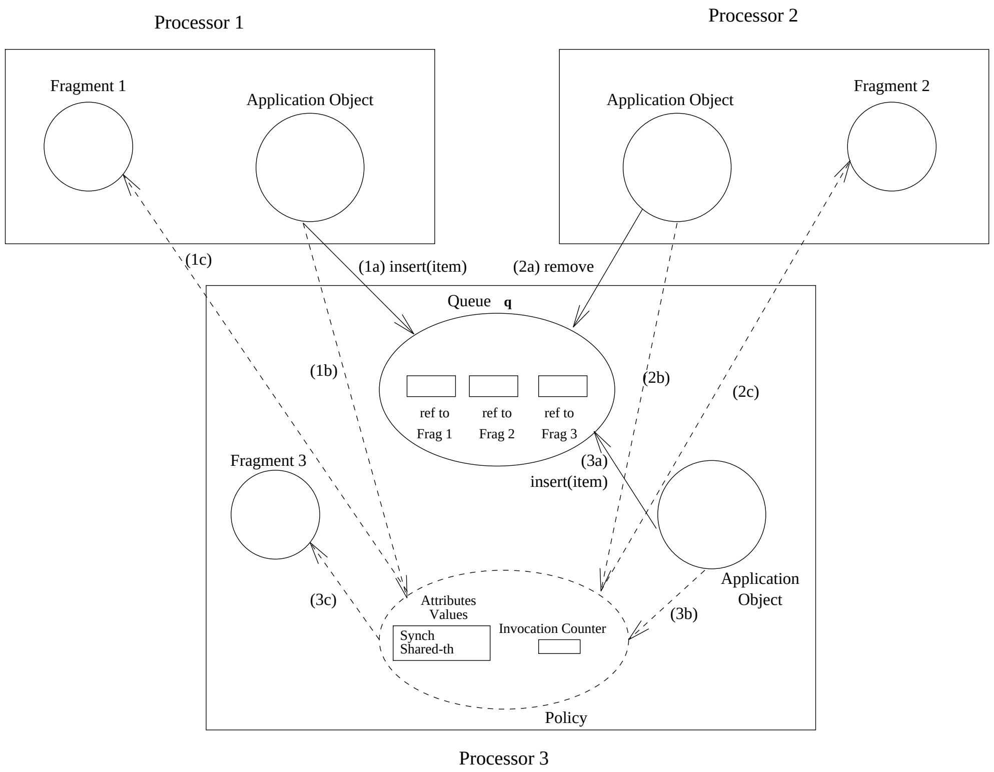

Figure 3: Invocation of a distributed queue using a single copy of a policy object

Figure 4 depicts the '`replicated`' queue for the same situation as shown in Figure 3. A local copy of *Queue q* works as a read-only copy of the 'main' `q` which is used in the previous, centralized design; it locally makes available all information that used to be contained in the 'main' q. This also holds for the policy objects with their replicated attributes and state. This implies the need for policy-specific implementations of any required global state (e.g., the operation counters used in some of the queue orderings), and it implies that different policies may use different attribute values. In the case of the distributed queue, this means that different fragments may behave in distinguished ways. This is desirable since application 'parts' can be in different states and may therefore require different behaviors from fragments. Of course, for general queue orderings, multiple replicated policy objects may have to interact, as is the case for protocols like 'shared-kth' where remote invocations are issued to share k^{th} values among fragments.

3.4 Discussion of CTK Abstractions and Design

We have added attributes to object interfaces as a low cost means for adaptation guidance; through them, clients can specify dynamic changes in behavior. Attribute values can be attached to object classes, object instances, state variables, operations, and object invocations. Attributes act like interaction gateways between functionality and the implementation of dynamic configuration aspects. They intend to represent information that the object designer wants to expose to the "external" objects responsible for configuration actions (performed by policy objects). They can also store

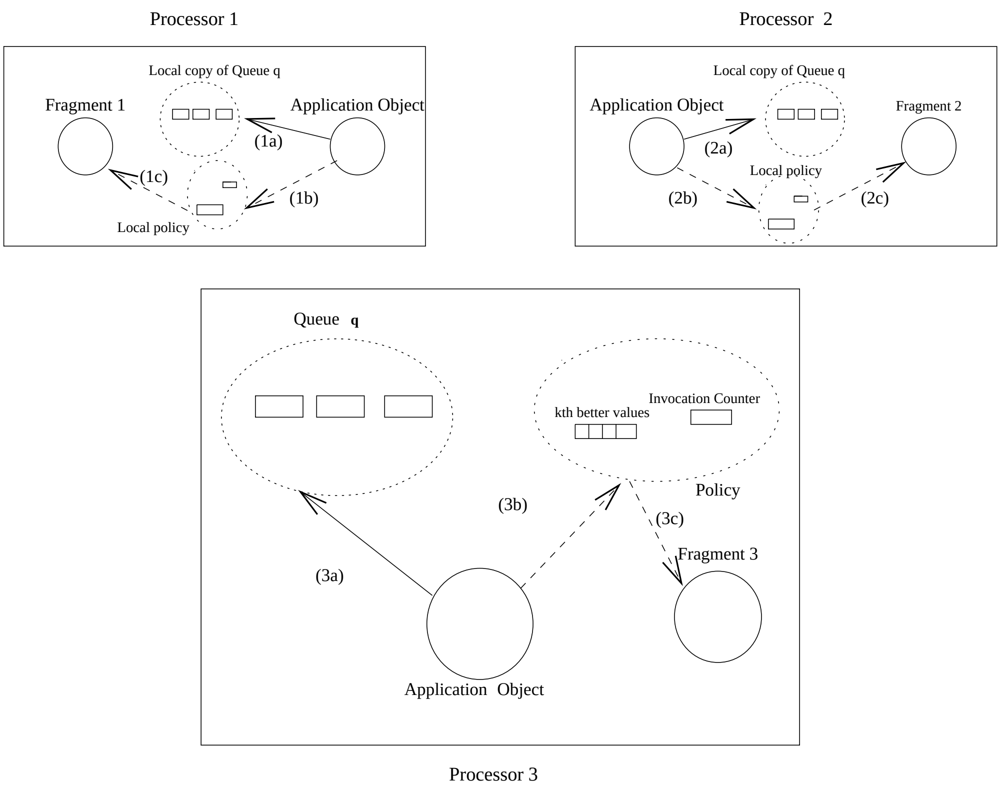

Figure 4: Invocations with local policy objects

information provided by the policy objects as a result of configuration decisions, so that functionality aspects can be changed accordingly.

Policies move configuration decisions and enactments away from the basic functionality, thereby allowing a disciplined variation among alternative implementation approaches. Policies use attributes values defined in the base object for decisions and actions, but they also work on extra-information that they explicitly gather, without depending on the base object functionality and implementation. This means that the attributes for customization may be defined along with each class, but also may be encapsulated in the policy. In situations where the dynamic configuration addresses issues orthogonal to the basic functionality implementation (for example, monitoring the operations performed on the object or attaining persistence by external storing of every object change), there is no explicit coupling between object and policy definitions, allowing complete reuse of the policies.

CTK allows for an association between objects and policies more general than the usual object/meta-object association. Since a single policy may be associated with a collection of objects, it can easily make configuration decisions and actions based on information that is not available to a single object. Invocations on any of the objects associated to the policy may trigger attribute values changes on the other objects. The implementation of these other objects may use the new values in order to proceed differently with their computation. CTK also supports the association of multiple policies with a single object, providing for separation of concerns. As discussed in Section 4.4, in some cases CTK can achieve this separation without additional execution overhead.

CTK objects can be created and manipulated using library routines only. Nonetheless, the use of our language support for CTK objects greatly simplifies application composition and documents the alternatives for configuration.

4 Experimentation with CTK Objects

4.1 CTK Implementation and Structure

In order to present and explain experimental results, we first describe some relevant detail about the multiprocessor implementation of CTK. The structure and implementation of the Configuration Toolkit are depicted in Figure 5 as consisting of three components: (1) configurable threads, which is the portable Cthreads package underlying CTK, (2) built-in object types and its support for attributes and policies, and (3) the various *policies* and *attributes* implemented for the application programs built with CTK and the hardware platforms for which CTK is optimized.

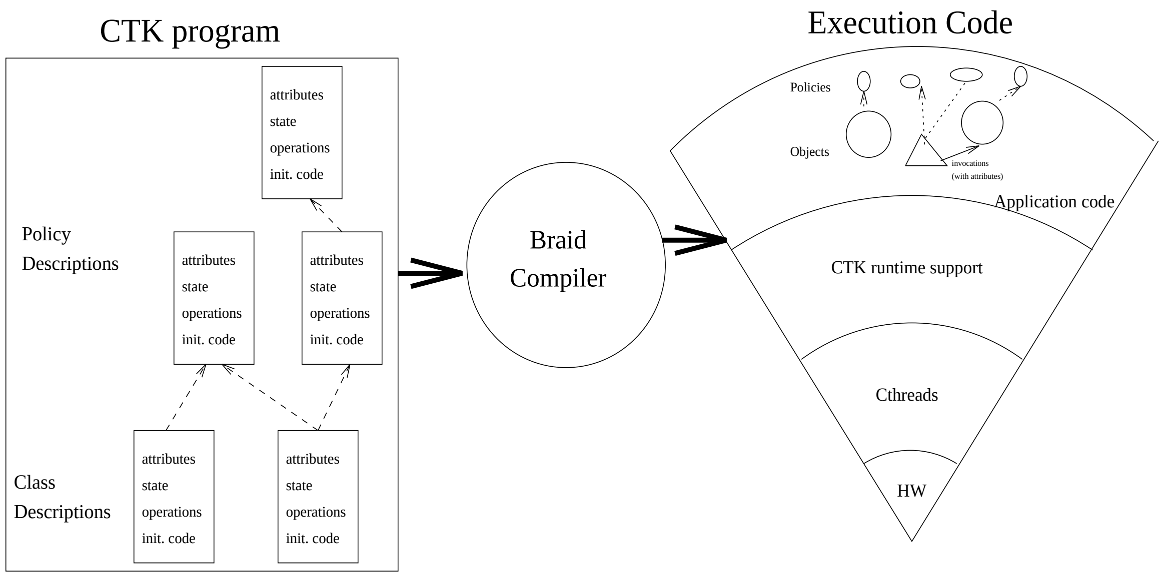

Figure 5: Structure of the Configuration Toolkit (CTK)

Configurable Cthreads is the partially machine dependent component[36] that implements the basic abstractions used by the remainder of CTK: *execution threads*, *virtual memory regions*, *synchronization primitives*, *monitoring support* (for capture of parallel program and CTK state), and a limited number of *basic attributes* for the configuration of threads-level abstractions, such as synchronization primitives and low-level scheduling. The Cthreads package is relevant to this paper in that it provides the lower level facilities utilized by the CTK library. Since Cthreads performance is much like that of other user-level threads packages, this paper does not explicitly discuss this topic, nor does it focus on the performance-relevant interactions of Cthreads with CTK.

CTK runtime support is comprised of the basic objects (ADT, TADT, etc.) offered by the library and the extensions targeting the KSR and SGI machines as well as targeting the specific applications investigated in this paper. These extensions include support for monitoring, persistence, replication, and application specific consistency policy for counters.

We next present some of the basic costs of CTK abstractions, followed by the evaluation of specific configurable objects constructed with CTK and utilizing the various extensions that support these objects. For the experiments on the KSR and SGI machines, application programs have control over the mapping of CTK abstractions and of application-level objects to the nodes and memory of the underlying parallel machines. Such memory and thread binding functionality is commonly available for modern parallel machines and operating systems.

4.2 Performance of the Configurable Distributed Queue

Summary of results. Two important results are described in this section. First, despite the apparent complexity of CTK's configuration support, the efficiency of its basic abstractions' implementation enables significant performance gains derived from the configuration of selected object attributes (in this case, the 'order' attribute in the TSP application's 'distributed queue' object). Second, one important characteristic of attributes and policies for high performance programming is their utility for exploring and experimenting with an application's performance behavior. In this example, for instance, it is useful to explore different 'order' settings experimentally, because it is impossible to determine statically the appropriate protocol for each specific TSP problem. In effect, it is the 'order' protocol which determines how successful the TSP application is in selecting 'good' subproblems for expansion and in avoiding the expansion of less useful subproblems.

The experimental results which appear below explore the effects of alternative queue orderings on the total time spent executing queue code vs. the application's total execution time. Concerning queue execution times:

- the "weak-order" queue has the least total execution time since there is almost no communication between different queue fragments (remote fragments are accessed only if there are requests for removals on empty local fragments);
- a totally ordered distributed queue requires that all fragments are locked on each removal, which makes this implementation inefficient for larger numbers of removals; and
- the other three ordering strategies ("give-second-best", "get-second-best", and "shared-kth") exhibit execution times that are proportional to the amounts of global communication (i.e., numbers and rates of queue elements shipped to neighbors).

However, while queue execution times are proportional to the communication amounts implied by orderings, they are inversely proportional to the 'qualities' of queue orderings implemented by each strategy. In other words, slower distributed queue implementations offer better qualities of element orderings and therefore lead to improved performance for the TSP application.

CTK basic costs. The costs of using CTK's mechanisms are sufficiently low (e.g., 15 μseconds for a local ADT object on the KSR-2 machine) to enable performance gains by their use. This cost compares favorably with the latencies of local procedure calls on the KSR machine (which we measured as 9 μseconds) and on the SGI (which we measured as 7 μseconds).

More importantly, CTK has been implemented such that the costs of using its mechanisms are proportional to their degree of use. For instance, while an ADT object is efficiently invoked, invocations of more complex objects like TADTs and Tasks cost more, on both the KSR and SGI machines[4]. As shown later in Table 7, more complex policies and additional invocation attributes typically increase invocation costs, but only when they are actually used. As a result, programs 'pay for' their use of more complex policies only when such usage is justified by application needs.

object type	*local object*		*remote object*	
	KSR	SGI	KSR	SGI
ADT invoke	15	33	23	23
TADT invoke	32	27	34	27
Task invoke	35	33	35	33
Monitor invoke	36	37	36	37

Table 1: Basic invocation costs in μseconds

The measurements in Table 1 are derived from runs on a 64-node KSR-2 supercomputer and on a 4-processor SGI multiprocessor. For brevity, we elide measurements of previous prototypes of CTK on other Unix machines, like the Sequent Symmetry, Sun3's, Sun386's, and measurements of its implementation as a native real-time kernel on the GP1000 BBN Butterfly multiprocessor[22]. The KSR-2 machine is a NUMA (non-uniform memory access) shared memory, cache-only architecture in which each node consists of a 64-bit processor with 32 Mbytes of main memory used as a local cache. CPU clock speed is 40 MHz, with peak performance of 40 Mflops per node. The parallel programming model implemented by the KSR's OSF Unix operating system is one of kernel-level threads, but CTK's implementation uses the user-level Cthreads package described in [35] in which user-level threads may be mapped one-to-one to kernel threads, thereby facilitating a unique mapping of user threads to processors.

The SGI machine is a symmetric multiprocessor with faster processors, memory, and interconnect than the KSR machine. It offers a computation/communication speed ratio not too dissimilar from that of the KSR machine. As with the KSR, user-level threads may be bound to the SGI's formulation of kernel-level threads, which in turn may be bound to processors. However, in contrast to the KSR machine, all memory is uniformly accessible, so that the super-linear speedup sometimes attained with KSR applications does not occur for this machine. Differences in the basic costs of CTK mechanisms across both machines are due to differences in machine architectures, processor speeds, and memory and bus speeds.

From the basic costs of CTK abstractions presented in Table 1 we may derive limits concerning the granularity of object usage in CTK programs. For the TSP application and for branch and bound codes in general, these limitations are acceptable, because even small grain codes typically require computation times of at least 100 μseconds to solve individual subproblems prior to branching and generating new subproblems for queue entry or consuming new subproblems by removing them from the queue. Similarly, since the overheads of CTK objects compare favorably with those of thread context switch times, even relatively small grain threads would be successfully scheduled with a CTK-constructed distributed queue.

[4]The measurements in Table 1 utilize a non-optimized implementation of CTK for the SGI machine. For the TADT object type, the values in the table do not include thread forking costs, which approximately double TADT invocation delays.

Experiment description. In the experiments described next, detailed evaluations of the relationships between ordering strategies' overheads vs. the effects of ordering qualities are accomplished by monitoring the distributed queue object and by subsequently processing the generated trace files in comparison to a simulated totally ordered queue. Namely, we assess comparative ordering qualities by inspection of the ranks of all elements being inserted and removed from the distributed global queue, in comparison to the simulated insertions and removals on the totally ordered queue. Using the following notations,'quality' is computed as:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{f - r_i + 1}{f}$$

where:

- n is the number of 'remove' operations, with $item_i$ denoting the element returned by the i^{th} removal;
- f is the number of fragments;
- q_i is the list of elements in the distributed queue at the time at which $remove_i$ is executed, with the elements sorted by priority; and
- r_i is the rank of element $item_i$ in q_i.

This measure of the effective quality of each ordering scheme quantifies how successfully a protocol coordinating access to a collection of queue fragments implements the totally ordered (centralized) queue abstraction. The idea is to compute how closely, on the average, each execution approximates a centralized queue's behavior. For instance, if a distributed queue implementation actually returns the 'best' element for each removal (ie., total order is attained), r_i would be 1 for every i, and the formula would compute $(f/f + \ldots + f/f)1/n = 1$, indicating 100% quality for this queue implementation. As another example, if the profile of the queue's operations for each node is *(insert, insert, remove, insert, remove, insert, ...)* for a queue composed of two fragments, and if removals return the globally best available elements in half of all operations and the second best for the other half, then the sum would indicate a total quality of 75%, which is simply the average of the $n/2$ occasions where the element returned is 100% 'good' and of the $n/2$ occasions where the answer is 50% off (ie, the returned second element has two possibilities). If this same pattern had occured with 10 fragments, then our quality measure would have computed 95% total quality and thereby indicated that the elements returned were very close to the best solution.

In order to separate the TSP program's from the queue's behavior, we perform experiments with the TSP application and with a workload generator:

1. The TSP application runs with all alternative ordering strategies. In these runs, nodes with 'good' priority values are more likely to keep generating good values, and many insertions are performed in the initial part of the execution, whereas the frequency of removals increases as the application program's execution progresses and peaks toward the end of each run.

2. A synthetic workload generator performs queue insertions and removals using uniform distributions for the qualities of elements being inserted into the queue and for the probability of performing insertions vs. removals. In comparison to the actual TSP application's behavior, this experiment makes it more difficult to realize gains from load balancing. As a result, a positive pattern of tradeoffs between queue execution time and order protocol quality would be even more advantageous under a TSP execution.

Results attained with the workload generator. One interesting insight from the experiments described next is the fact that a partial order scheme with a fragmented queue has a behavior close to that of a totally ordered centralized queue implementation, while at the same time, offering substantial benefits in terms of queue execution times. These results hold for experiments with the TSP application; they also hold for measurements attained with a workload generator when queue accesses and element qualities are drawn from uniform distributions. As expected, with a small number of processors, the difference in performance between the centralized behavior – total order – and the fully distributed behavior – weak order – is small (as shown in Table 2). As the number of processors increases, the quality/execution time tradeoffs offered by partially ordered queues have significant effects on overall program performance.

Note that the quality values for 'total-order' are not the expected 100% (Table 3). This is because our analysis algorithm serializes the operations performed by the parallel execution using time stamps; two removal operations having the same time stamp will be serialized in the parallel execution by the lock acquisition in a way not captured by traces. Clock resolution and clock synchronization among processors are other factors impeding the exact reproduction of the parallel execution history.

Detailed measurements appear in Tables 2, 3, and 4. These measurements are attained with the workload generator, using uniform distributions for element quality values and for queue access operations. Experiment runs are performed for different numbers of processors and multiple values for the attributes 'order', 'how_often', and 'k', with a total of about 600 experiments for the synthetic workloads and about 100 for the TSP application. In the give-second-best, get-second-best, and shared-kth protocols, the attribute value for 'how-often' is 5; for the shared-kth protocol 'k' has the value 3. We denote by f the number of fragments (and processors) running parts of the application. Each simulation has approximately $80,000$ operations that are evenly divided among the processors; execution times and quality values in the tables are the averages of measurements from five program runs.

The numbers in Tables 2 and 4 refer to the implementation where both the queue and its policy (i.e., information about its fragmentation and possible configuration strategies) are replicated across processors. On the KSR machine, policy replication improves execution time by roughly 8% in most cases. More importantly, these experiments conducted with the workload generator clearly show the tradeoff in the quality of element ordering vs. queue execution time. Namely, while the 'weakly' ordered queue has the least execution time, it also offers the least quality. Similarly, the totally ordered queue is the slowest, but its element ordering is the best. All other orderings' behavior falls between both of these extremes. Moreover, these results hold for both the KSR machine and for symmetric multiprocessors, like the SGI multiprocessor, as evident from the results in Table 4. It is likely that they will also hold for other NUMA machines like the Stanford DASH multiprocessor and multiple SGI SMP subsystems coupled with high performance memory interconnects like the Origin series of SGI machines.

The synthetic workload tends to generate queues that remain small throughout the program's execution. Therefore, it is reasonable to expect a higher incidence of empty fragments, which should cause the protocol "get-second-best" to run slower than "give-second-best". In the experiments using the SGI machine (Table 4), "give-second-best" always offers better results than "get-second-best". In the KSR experiments, both protocols have similar execution times (as illustrated in Table 2).

ordering protocol	$f = 2$	$f = 4$	$f = 8$	$f = 10$	$f = 16$	$f = 32$
weak	14.386	8.776	4.015	3.981	2.963	2.771
get-second-best	16.44	9.744	4.477	4.271	4.210	3.900
give-second-best	16.442	9.639	4.869	4.019	3.785	3.016
share-kth	16.237	9.781	4.835	3.985	3.173	2.603
total	20.8982	12.796	7.1663	6.023	4.8627	4.0134

Table 2: Execution times for workloads generated from uniform distributions (in secs.) (KSR-2)

ordering protocol	$f = 2$	$f = 4$	$f = 8$	$f = 10$	$f = 16$	$f = 32$
weak	84	77	61	57	51	48
get-second-best	87	83	83	79	78	78
give-second-best	87	83	75	73	70	73
share-kth	88	86	86	84	85	80
total	100	99	97	98	100	99

Table 3: Quality measures for workloads using uniform distributions (KSR-2)

Figures 6 and 7 depict the data from Tables 2 and 3 graphically. These graphs clearly distinguish the tradeoffs among alternative element distribution policies with respect to cost vs. ordering qualities. We also conclude from these results that it is important to experiment with alternative partial queue implementations, since the performance gains attainable with these implementations are neither apparent nor easily anticipated.

Results attained with the TSP application. Experimental outcomes of runs with the TSP application and alternative queue implementations are depicted in Tables 5 and 6, and in Figure 8. The two tables list the execution times and quality measures obtained when using the configurable fragmented queue with the TSP code. In these experiments, the TSP program's branch-and-bound approach solves the 2-EUCLIDEAN TSP problem for 51 cities, as specified in TSPLIB[5]. The graph in Figure 8 repeats the measurements shown in Table 5. It is interesting to note that the centralized

[5]The Traveling salesman problem library(TSPLIB) provides researchers with a broad set of standard test problems[41].

ordering protocol	execution time			quality (%)		
	$f = 2$	$f = 3$	$f = 4$	$f = 2$	$f = 3$	$f = 4$
weak	11.29	9.01	6.03	87	80	78
get-second-best	13.35	12.62	10.09	89	83	79
give-second-best	13.01	12.54	9.98	89	83	80
share-kth	13.26	12.72	9.99	88	86	85
total	16.21	15.11	12.02	100	100	99

Table 4: Execution times for workloads using uniform distributions (in secs.) (SGI)

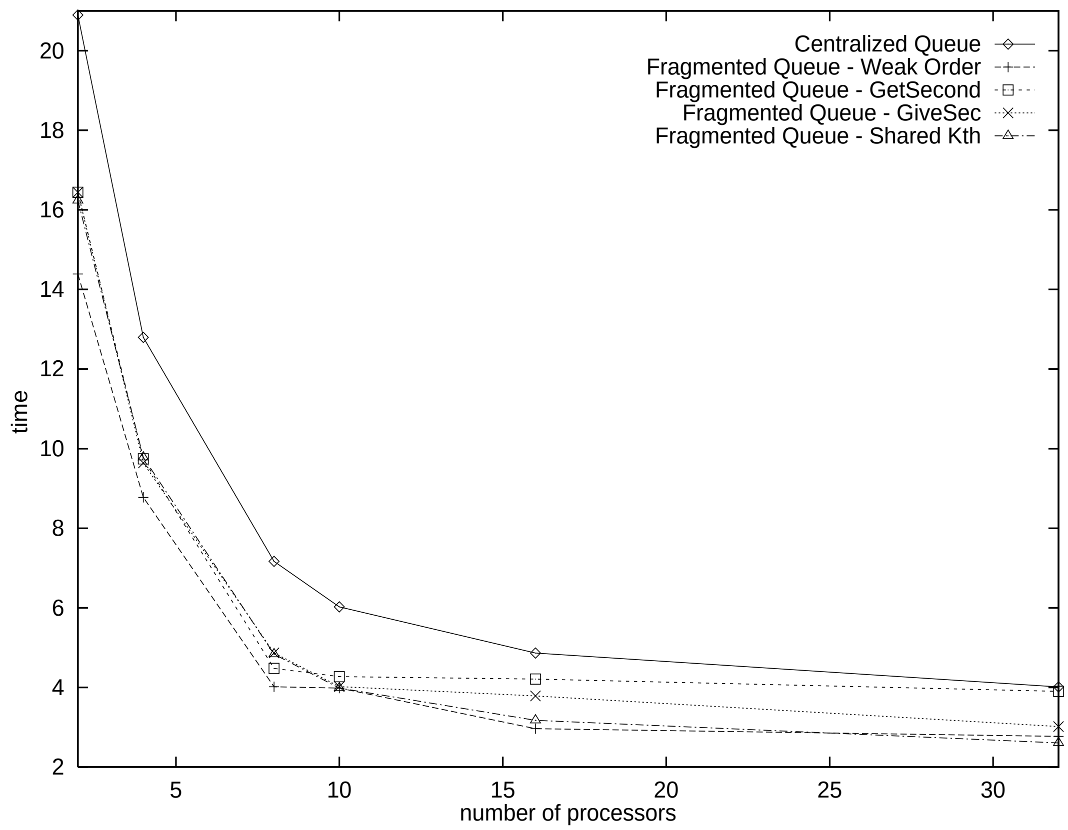

Figure 6: Execution times for workloads using uniform distributions (in secs.) (KSR-2)

queue outperforms the weakly ordered queue for smaller numbers of processors due to its superior queue quality. However, with an increasing number of processors, the additional communication costs overshadow the quality gains. Then, once the number of processors exceeds 10, the weakly ordered TSP execution starts to spend most of the CPU time working on suboptimal problems (optimal locally, but not globally), and the centralized queue results in TSP execution times that outperform those using the weakly ordered queue.

ordering protocol	$f = 2$	$f = 4$	$f = 8$	$f = 10$	$f = 16$	$f = 32$
weak	73.2	60.1	43.4	35.2	31.7	28.0
get-second-best	39.9	31.8	28.8	23.4	21.1	20.7
give-second-best	39.1	30.3	25.0	21.4	19.1	17.7
share-kth	34.5	27.9	18.8	15.1	13.3	12.7
total	67.4	51.3	45.4	39.3	29.2	25.0

Table 5: TSP execution times with configurable queue, for 51 cities (in secs.) (KSR-2)

Discussion of experimentation. This section has demonstrated the benefits derived from object configuration implemented with CTK's policies and attributes. In part, such benefits are due to the

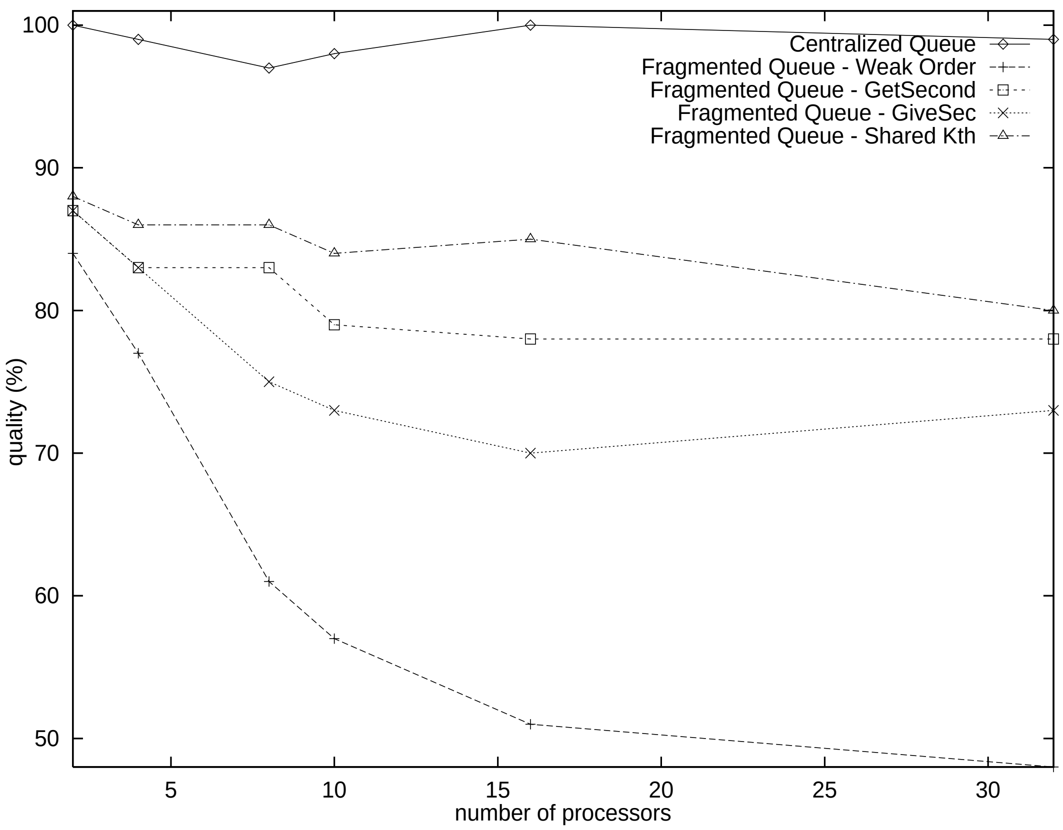

Figure 7: Quality measures for workloads using uniform distributions (KSR-2)

efficient and 'pay as needed' implementation of CTK's configuration mechanisms. More importantly, benefits from object configuration result from the use of attributes and policies that can customize an abstraction's behavior to the needs of a particular application program. Since such policies' functionalities may be varied separately from basic object behaviors, it is straightforward to experiment with alternative policies and policy implementations, to develop appropriate attributes and policies experimentally, and to add or remove certain object behaviors. We next describe additional ways in which separated policies and attributes may be used to configure CTK objects.

4.3 Using Policies for Object Extension

Multiple policies: a monitored, persistent, distributed queue. Section 4.2 demonstrates the utility of CTK for performance improvement, by implementing and evaluating alternative policies and attributes for a distributed queue abstraction. More generally, such customization of abstractions — in order to adapt to match application needs, for example — may involve the configuration of different and multiple characteristics of object behavior. For instance, in the experiments with the distributed queue, the queue abstraction itself had to be specialized in two ways:

1. queue accesses and queue quality had to be *monitored* so that queue behavior could be evaluated for each of the five ordering schemes; and

ordering protocol	$f = 2$	$f = 4$	$f = 8$	$f = 10$	$f = 16$	$f = 32$
weak	84	77	61	57	51	48
get-second-best	88	86	86	84	85	80
give-second-best	87	83	75	73	70	73
share-kth	87	83	83	79	78	78
total	100	99	97	98	100	99

Table 6: Queue quality measures for execution of TSP, for 51 cities (KSR-2)

2. all queue accesses as well as selected queue state had to be made *persistent*, (1) for postmortem evaluation of queue behavior, and (2) because such persistence facilitated experiments with large TSP problems on the somewhat unstable KSR hardware. With persistence, after a crash, the application could be restarted from the last queue state resident on the disk.

In our approach to object configuration, each policy encapsulates one specific aspect of configuration. For a monitored, persistent, and distributed queue, this implies the association of multiple policies with the queue object. Each such policy/object combination implements a certain specialization, customization, or extension of the abstraction. Two extensions of object functionality via additional policies – persistence and monitoring – are described next, followed by an experimental evaluation of the costs of multiple object extensions via such 'stacked' policies.

Persistent objects. The implementation of object persistence described next does not attempt to explore alternative or efficient ways of offering persistent object behavior. Instead, the persistence policy's implementation with CTK simply demonstrates how to modify the behavior of an object in ways not related to its basic functionality:

1. the association of a new policy with an otherwise fully functional object enhances the object's functionality;
2. the extended functionality is made accessible via attributes specified as part of the object's changed system view; and
3. the new policy may itself be configured (via additional attributes) in order to specialize its implementation of persistence to match the underlying operating system or hardware resources. For example, attributes permit the selection of storage media (e.g, the local file system or a remote data server) for persistence data.

For brevity, details of the persistence policy's implementation are elided in this paper (see [49]). Instead, we next discuss the implications of two alternative persistence policy realizations for the TSP application:

- *Checkpointing:*
  Periodically, all of the information about the state and the attribute values of queue fragments is placed into checkpoint files. An application-specific restart routine recovers this state and additional information about the application (e.g., the minimum tour value, the data structure being used to generate the new instances to be added to the search space, etc). The efficient implementation of such checkpointing performed as part of this research does not use barrier synchronization to enforce some notion of global queue consistency. The resulting performance

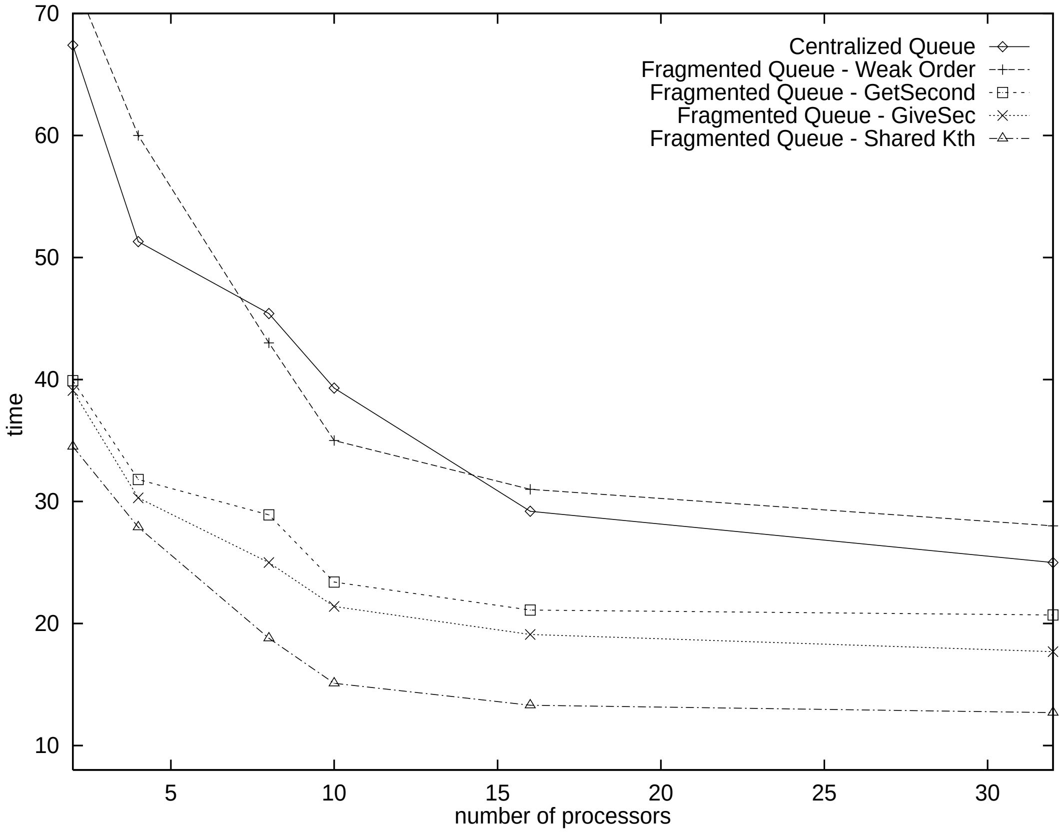

Figure 8: TSP execution times for 51 cities (in secs.) (KSR-2)

gains are attained at a price: checkpointing alters the distributed queue's ordering policy. Namely, while the checkpointing process is being carried out on a certain queue fragment, insertion and removal operations on other fragments continue to run.

- *Logging operations:*
  A 'logging' persistence policy commits to storage every invocation's argument values prior to actually carrying out this invocation. However, its use with the TSP application has high costs due to the application's rapid generation and expansion of many different subproblems. In comparison, its use with the branch-and-bound solutions to the processor mapping problem described in [43] is appropriate, since that problem's relatively small number of expansions each require substantive amounts of processing.

Despite its prohibitive costs, our implementation of the logging policy is interesting since it addresses how logging may be performed for internally fragmented objects. In the TSP application, a logging policy records argument values that each consist of a list of edges selected for expansion. In order to restart TSP, the log file containing edge lists is consumed like a script that replays the application. Interestingly, when applied to the fragmented queue object, this persistence policy 'sees' only queue operations initiated by the application, and it does not perceive any inter-fragment operations executed 'within' the distributed object itself. Therefore, if the execution of a removal operation on processor of fragment q_i returns an element e donated by the fragment q_j $(i \neq j)$, there may be no indication in the log that e has been removed from q_j and has been inserted into q_i.

Interactions between different object fragments complicate the implementation of a logging policy for the queue object. Specifically, in order to guarantee that an element e (representing a node in the search space) will not be processed twice, every time e is returned without indication of having been inserted, it is necessary to identify the queue fragment for which e has been logged as inserted. This requires that *identifiers* have to be assigned that uniquely identify the generated 'branch' nodes, and that these identifiers also uniquely denote the queue fragment that contains the element in question.

The discussion of logging in the previous paragraph assumes that operations are logged at the application interface of the fragmented object. An alternative approach to logging might look at the operations on each queue fragment. The advantage would be that all insertions and removals (including the ones derived from actions to balance the queue) would appear in the log, and therefore the log for a queue fragment would be an exact image of the queue. The approach's disadvantage is that obtaining a snapshot of available subproblems would require that all policies cooperate in achieving global synchronization.

Monitored objects. The monitoring of CTK objects is implemented via a policy that internally utilizes the online monitoring mechanisms provided by the *Falcon* project [23]. Using Falcon's basic mechanisms of sensors, local monitoring threads, and remote monitors, CTK's monitoring policy implements the following useful functionality:

- it timestamps the initiation and the conclusion of each invocation, while 'subtracting' the effects of policy executions associated with such invocations (e.g., abstraction-specific policies or policies implementing diverse invocation semantics);
- it provides information about when policies are activated as part of invocation executions, and it distinguishes replicated from non-replicated policies; and
- it captures the values of attributes passed via invocations. Note that such values are not typically available to the users of certain abstractions, since such users are not given access to the abstractions' and their policies' implementations.

4.4 Compiler Support for Optimizing the Use of Multiple Policies

Figure 9 shows the relationship between objects and policies for the monitored, persistent, distributed queue. With CTK, this complex abstraction is constructed incrementally. First, the 'Distributed Queue' abstraction is defined, followed by definition of the 'Persistent Distributed Queue'. Finally, the 'Monitored' complex object is specified. When this complex abstraction is invoked, the invocation is handled by three different policies, before reaching the appropriate queue fragment, as shown in the edges labeled (1) to (4) in Figure 9.

A straightforward implementation of multiple policies' association with an object simply permits each policy to interpret each object invocation using the appropriate attributes, then forwards the invocation to the next policy. To avoid such repeated invocations (and the costs incurred on parallel platforms), CTK's compiler support simply collapses such 'stacked' policies into a single unit, in the order specified by the clauses of policies in the object definition. The policy implementor continues to perceive the collapsed policies as separate entities. However, since the Braid compiler does not have any semantic knowledge of policies and their interactions, it is up to the implementor to identify

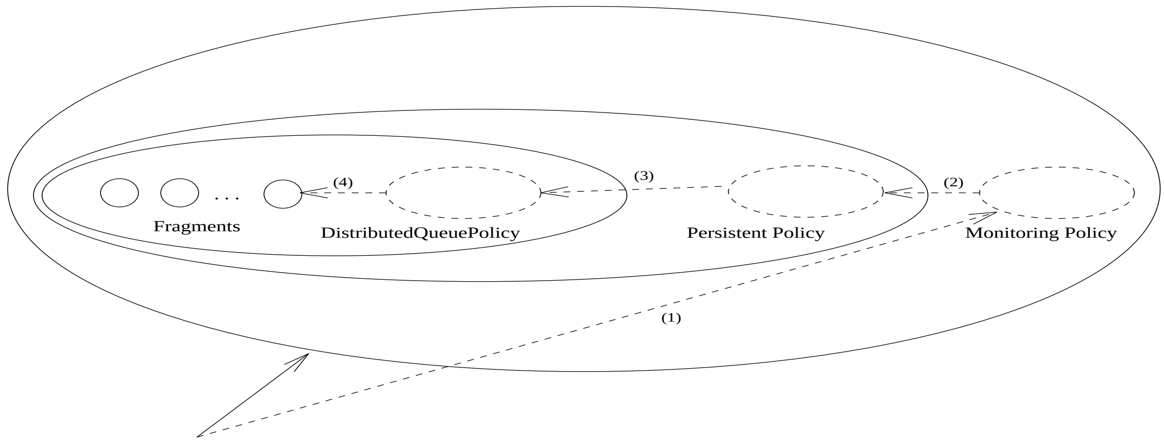

Figure 9: Monitored Persistent Distributed Queue object

and deal with potential policy interactions when 'stacking' them. One manner of dealing with such interactions supported by Braid's policy stacking tool is for one policy to change a subsequent policy's behavior via dynamic attribute specification. For example, for the fragmented queue, the *Persistence* policy temporarily changes the value of the attribute 'order' passed to the *DistributedQueuePolicy* to the value "weak". This prevents load balancing actions from being carried for the duration of the *Persistent* policy interaction with the queue object.

The performance effects of 'flattening' hierarchically stacked polices are depicted in Table 7. Two insights into CTK's implementation may be derived from these measurements: (1) invocation costs are directly proportional to the number of attributes passed to an object's policy (e.g., compare the 'Distributed' to the 'Persistent Distributed' queue), (2) the association of multiple policies with an object increases invocation costs beyond the costs incurred by each policy (e.g., compare the 'Persistent' with the 'Monitored' with the 'Persistent Monitored' queues), and (3) compiler actions to reduce the number of policies traversed for each invocation can improve performance substantially, as evident from comparing the last two entries of the table.

object	*nb of policies*	*nb of attributes*	*basic invoc. cost*
Fragment	0	0	15
Distributed Queue	1	2	30
Persistent Distr. Queue	2	3	37
Monitored Distr. Queue	2	3	40
Pers. Monit. Distr. Queue	3	4	54
Pers. Monit. Distr. Queue (flattened)	3	4	38

Table 7: Invocation costs in μseconds for complex objects and policies

4.5 Discussion

It is unlikely that programmers can design *a priori* solutions for how to configure objects like fragmented queues for dynamic applications such as the TSP code, because they would have to anticipate the application's runtime behavior. Such problems are addressed by CTK's lightweight configuration mechanisms, which permits programmers to explore alternative protocols for such shared abstractions. Specifically, CTK provides support for (1) pursuing a variety of configuration possibilities and (2) setting up an environment in which experiments are easily conducted and evaluated. For example, the data presented in this section results from employing the best choices for attributes and policy behavior found during our extensive experiments with each queue ordering protocol. Experiments with the TSP application are limited to the configurations proved useful during the queue experiments. When conducting such experiments, ease of programming is enhanced by the separation of performance issues and functional extensions (persistence, monitoring) from basic object functionality, and by the CTK runtime system's ability to configure the underlying Cthreads package (*e.g.,* adding monitoring). These attributes of CTK also facilitate the automation of experiments' analysis procedures. Last, the CTK compiler's support for integrating policies into a single component and by replicating objects/policies are crucial for achieving low invocation overheads.

5 Conclusions and Future Research

The *Configuration Toolkit* (CTK) described and evaluated in this paper is an object-based parallel programming library. CTK provides an efficient basis for building configurable high performance programs for multiprocessor engines. CTK has been used to implement parallel applications, and it has provided a basis for building a real-time operating system as a native kernel, on target platforms ranging from small scale symmetric multiprocessors to parallel supercomputers. CTK permits object developers to separate an object's basic functionality from its implementation characteristics determining its performance or reliability. This separation allows developers to experiment with alternative object implementations, by variation of policies associated with objects. In addition, specific implementation characteristics may be exposed, using attributes manipulated by policies. Since attributes may be interpreted at runtime, they also permit the online configuration of object performance or reliability to match current application needs.

In this paper, CTK is shown useful for implementing sample complex object behaviors required for high performance programs running on parallel machines. Such behaviors are specified with simple language constructs, and they are implemented without knowledge of the internal implementation of CTK's runtime support. Moreover, performance improvements may be realized using any number of configuration techniques, including: (1) the dynamic interposition of configuration code between object invoker and implementation, (2) the parameterization of selected object characteristics, and (3) object fragmentation or replication and the efficient runtime maintenance of fragments or replicas.

This paper demonstrates that performance improvements derived from the configuration of object attributes range from 10%-50%, for a variety of parallel application programs and abstractions, on a SGI symmetric multiprocessor and on a KSR shared memory supercomputer. For larger-scale parallel applications, we expect to achieve cumulative performance gains approaching or exceeding 100%. Similarly, in our current work with the runtime configuration of objects on distributed platforms[49, 50], we expect to realize substantial performance gains by configuring applications simultaneously at several levels of abstraction in the distributed program and in the underlying operating systems and networks: (1) in the application itself, (2) in the object transport substrate[54], and (3) at the protocol level. Our previous work has demonstrated the performance gains from configuration at each of these levels[27, 37], but we have not yet been able to show gains by simultaneous configuration at multiple levels of abstraction.

The mechanisms for configuration in CTK may also be used for the customization of operating system kernels, by specializing and/or modifying existing CTK-constructed kernel abstractions. 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 the CTK library, resulting in a complex real-time operating system kernel [22]. We have also experimented with a second extension of CTK supporting configurable communication protocols[30]. In the future, we expect to apply the principles demonstrated with CTK toward the development of customization and specialization facilities for both object- and agent-based CORBA-compliant middleware[2].

The actual extensions of object functionality described in this paper concern persistence and monitoring. In general, the extensions considered in our research concern services typically provided (in some non-configurable fashion) by operating systems. By permitting such services to be customized for certain applications, gains in performance and reliability may be realized, and applications may be tailored to the characteristics of specific underlying hardware platforms.

Our future research has two aims: (1) to generalize the notion of policies and attributes to facilitate their use in future high performance distributed and parallel applications, and (2) to utilize the configuration mechanisms presented in this paper to develop new technologies for online program configuration. Concerning (2), our group has been investigating the *interactive steering* of high performance programs, to realize gains in performance and functionality[52]. The essential idea of this technology is to give users the ability to configure remotely (if needed) selected attributes of their application-level objects while programs are running, using formulations of configurable object abstractions in distributed systems much like those implemented by CTK for multiprocessors[50, 16]. Concerning (1), our recent work on object-agent systems attempts to generalize the notions of policies and attributes to associate configuration functionality that is applied to both CORBA-based object and Java-based agent implementations[2].

Acknowledgments

We would like to thank Ahmed Gheith (now at Trilogy Corporation in Austin, TX), since CTK uses the basic ideas described in his PhD thesis and also uses his system's support for multi-weight objects. Weiming Gu and Jeffrey Vetter are responsible for some of the monitoring and steering support utilized in CTK. In addition, we would like to thank the reviewers for their extensive comments on this paper.

References

[1] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for Change: A Reconfiguration Language for Distributed Systems. In *Proc. of the Second International Workshop in Configurable Distributed Systems*, pages 15–22. IEEE Computer Society Press, May 1994.

[2] M. Ahamad, R. Das, and K. Schwan. Integrating Object and Agent Technologies for High-end Collaborative Applications . http:/www.cc.gatech.edu/fac/Mustaque.Ahamad/objent.html, 1998.

[3] S. Ben Hassen, H. Bal, and A. Tanenbaum. Hawk: a Runtime System for Partitioned Objects. *Journal of Parallel Algorithms and Applications*, 12:205–230, 1997.

[4] T. Bihari and K. Schwan. Dynamic Adaptation of Real-Time Software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[5] K. P. Birman and et.al. Implementing Fault-Tolerant Distributed Objects. *IEEE Transactions on Software Engineering*, pages 502–508, June 1985.

[6] R. D. Blumofe and D. Papadopoulos. The Performance of Work Stealing in Multiprogrammed Environments. Technical report, University of Texas at Austin, Department of Computer Science, May 1998.

[7] R. Campbell, V. Russo, and G. Johnson. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[8] *Proc. of the Fourth International Workshop in Configurable Distributed Systems*. IEEE Computer Society, May 1998.

[9] E. Chaves Jr., P. Das, T. LeBlanc, B. Marsh, and M. Scott. Kernel-Kernel Communication in a Shared-Memory Multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–192, May 1993.

[10] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133. ACM Press, December 1993.

[11] S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-level Architecture. In O. Nierstraz, editor, *7th European Conference on Object-Oriented Programming*, volume 707 of *LNCS (Lecture Notes in Computer Science)*, pages 482–501. Springer Verlag, July 1993.

[12] C. Clemencon, B. Mukherjee, and K. Schwan. Distributed Shared Abstractions (DSA) on Multiprocessors. *IEEE Transactions on Software Engineering*, 22(2):132–152, February 1996.

[13] G. Cox, W. M. Corwin, K. K. Lai, and F. J. Pollack. A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment. In *Proc. of the 8th ACM Symposium on Operating System Principles*, pages 44–53, Dec. 1981.

[14] S. Danforth and I. Forman. Reflections on Metaclass Programming in SOM. In *Proc. of OOPSLA'94*, pages 440–452. ACM Press, October 1994.

[15] P. Druschel, L. Peterson, and N. Hutchinson. Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 512–520, June 1992.

[16] G. Eisenhauer and K. Schwan. An Object-Based Infrastructure for Program Monitoring and Steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[17] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th Symposium on Operating Systems Principles*. ACM Press, December 1995.

[18] D. Ferrari, A. Banerjea, and H. Zhang. Network Support for Multimedia: A Discussion of the Tenet Approach. *Computer Networks and ISDN Systems*, 26(10), July 1994.

[19] P. G. S. Florissi and Y. Yemini. Management of Application Quality of Service. In *Proc. of the 5th IEEE International Workshop on Distributed Systems: Operations and Management*, 1994.

[20] B. Foote. Object-Oriented Reflection and Metalevel Architectures. *OOPS Messenger*, 5(2), April 1994.

[21] H. P. F. Forum. High Performance Fortran Language Specification, version 1.0. Technical Report CRPC-TR 92225, Rice University, 1996.

[22] A. Gheith and K. 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, Apr. 1993.

[23] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line Monitoring for Steering Parallel Programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[24] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A Flexible Base for Distributed Computing. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79. ACM Press, December 1993.

[25] P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and W. de Jonge. An Object Model for Flexible Distributed Systems. In *Proc. of ASCI'95, ASCI T.U. Delft*, pages 69–78, 1995.

[26] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[27] D. Ivan-Rosu and K. Schwan. Improving Protocol Performance by Dynamic Control of Communication Resources. In *Second International Conference on Engineering Complex Systems, Montreal*, pages 249–256. IEEE, October 1996. Outstanding paper award.

[28] M. Jones. Interposition agents: Transparently Interposing User Code at the System Interface. *Operating System Review*, 27(5):69–79, December 1993. Proc. 14th ACM Symp. on Operating Systems Principles.

[29] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[30] B. Lindgren, B. Krupczak, M. Ammar, and K. Schwan. An Architecture and Toolkit for Parallel and Configurable Protocols. In *Proceedings of the International Conference on Network Protocols (ICNP-93)*, September 1993.

[31] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages Can Be Implemented Efficiently. In *Proceedings of OOPSLA*, pages 127–145, October 1992.

[32] H. Masuhara, S. Matsuoka, and A. Yonezawa. An Object-Oriented Concurrent Reflective Language for Dynamic Resource Management in Highly Parallel Computing. IPSJ Notes, 94-PRG-18, 57-64, 1994.

[33] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), November 1995. (Also available in http:/www.cs.wisc.edu/paradyn/.

[34] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, and G. Zlokapa. Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel. In *Proc. of the IEEE Workshop on Real-Time Operating Systems and Software*, 1990.

[35] B. Mukherjee. A Portable and Reconfigurable Threads Package. In *Proc. of the Sun User Group Technical Conference*, pages 101–112, June 1991.

[36] B. Mukherjee and K. 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.

[37] B. Mukherjee, D. Silva, K. Schwan, and A. Gheith. KTK: Kernel Support for Configurable Objects and Invocations. *Distributed Systems Engineering Journal*, 1:259–270, 1994.

[38] A. Paepcke, editor. *Object-Oriented Programming – The CLOS Perspective.* MIT Press, 1993.

[39] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[40] R. Rashid, D. Julin, et al. Mach: A System Software Kernel. In *Proc. of the 34th IEEE Computer society International Conference (COMPCON 89)*, pages 176–178, February 1989.

[41] G. Reinelt. TSPLIB — A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[42] D. Revel, D. McNamee, D. Steere, and J. Walpole. Adaptive Prefetching for Device Independent File I/O. In *Proc. Multimedia Computing and Networking 1998 (MMCN98)*, 1998.

[43] D. I. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. In *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pages 320–329. IEEE, Dec. 1997.

[44] D. Schmidt. The ADAPTIVE Communication Environment. In *Proc. of the 11th Sun User Group Conference*, 1993.

[45] D. Schmidt, D. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4), April 1998.

[46] D. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale. Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1998.

[47] K. Schwan, B. Blake, W. Bo, and J. 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.

[48] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 198–204. IEEE, May 1986.

[49] D. M. Silva. *A Framework for Building Flexible Systems.* PhD thesis, Georgia Institute of Technology, 1997.

[50] D. M. Silva, K. Schwan, and G. Eisenhauer. Configurable Distributed Retrieval of Scientific Data. In *Proc. 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 120–127, Annapolis, USA, May 1998. IEEE Computer Society.

[51] A. Tucker and A. Gupta. Process Control and Scheduling Issues on a Network of Multiprocessors. In *Proc. of the 12th ACM Symposium on Operating System Principles*, pages 159–166, Dec. 1989.

[52] J. Vetter and K. Schwan. High Performance Computational Steering of Physical Simulations. In *International Parallel Processing Symposium (IPPS)*, Geneva, April 1997. IEEE.

[53] E. A. West and A. S. Grimshaw. Braid: Integrating Task and Data Parallelism. In *5th Symposium on the Frontiers of Massively Parallel Computing*, Washington, DC, February 1995.

[54] R. West, K. Schwan, I. Tacic, and M. Ahamad. Exploiting Temporal and Spatial Constraints on Distributed Shared Objects. In *Proc. 17th IEEE International Conference on Distributed Computing Systems*, 1997. (Also available is the full-length Technical Report: GIT-CC-96-36).

[55] W. A. Wulf, R. Levin, and S. R. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill Advanced Computer Science Series, 1981.

[56] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proc. of OOPSLA'92*, pages 414–434. ACM Press, October 1992.

[57] J. Zinky, D. E. Bakken, and R. Schantz. Architecture Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, January 1997. Also see http:/www.dist-systems.bbn.com/tech/QuO/.