A Framework for Building Complex Systems

A THESIS
Presented to
The Academic Faculty

by

Dilma Menezes da Silva

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
February 1997

Acknowledgments

I am *very* grateful to Professor Karsten Schwan for all the orientation, support, and encouragement he consistently gave me along these last six years. I really appreciate his patience during the long degree process. I would like also to thank the committee members: Mustaque Ahamad, Kishore Ramachandran, Raja Das, and Ajei Gopal.

I will never be able to express how important my friends in the lab were in providing a wonderful working environment. In particular, Greg Eisenhauer showed me that it is possible to be very productive *and* find time to help other people with their work. He is simply amazing as a project partner and friend. I could fill up pages with praise, and it would not be enough. Also, I would like to thank Ahmed Gheith, since this work uses some of the basic ideas in his PhD thesis.

I will be leaving Atlanta with much more than my degree. I carry in my heart the warm happiness of meeting people like Hernán Astudillo, JJ Tsai, Luis Carlos Santos, Abhijit Chadouri. I wish to have them in my life forever. I hope that Kaushik Gosh goes to visit me in Brazil many times. I will always remember the nice parties and brunches with Bodhi Mukherjee, Ram Kordale, Prince Kohli, Kiran

Panesar, Aman Singla, and many more interesting and nice people. I learned a lot from my roommate, Cristina Fernandes. She and her sister, Mary Carbinatto, will always feel like family. I am very glad that Georgia Tech housed so many nice Brazilian students: Sérgio Oliva, Carlos Cesnik, Lucio Flavio Pessoa, Tito Homem de Mello.

I received long distance support from many people; through daily e-mails we managed to participate in each other lives. I will not enumerate all the names, but I feel urged to thank Carlos Eduardo Ferreira and Marcelo G. Queiroz for the constant and loving presence in my mailbox.

I owe absolutely everything to my wonderful parents and siblings. *Thank you very very much.*

Finally, I would like to acknowledge the financial support I received during four years from *CNPq* (Centro de Desenvolvimento Científico e Tecnólogico) and *Universidade de São Paulo.*

Contents

List of Tables

List of Figures

Summary

Application requirements on operating system services and resources can vary widely. Moreover, such variability is often experienced at runtime, especially with complex distributed applications like multi-media systems and groupware systems. This suggests that middleware and underlying operating system services may derive significant performance improvements from the ability to adapt at runtime to their current execution environment. Similarly, for high performance parallel programs, flexibility in their design and an ability to be configured at runtime has almost become a requirement in the sense that levels of parallelism and the behaviors of scheduling and synchronization mechanisms may have to be varied to accommodate variations in program behavior or in the availability of computing and storage resources.

This thesis demonstrates that the use of configurable objects in multiprocessor environments can lead to substantive performance gains. The Configuration Toolkit (CTK) is a library for constructing object-based abstractions ranging from multiprocessor operating system services to user-level objects in parallel programs.

CTK is unique in its exploration of configuration issues: (1) it provides the developer with a programming model to express and explore configuration; (2) it offers runtime support for achieving on- and off-line adaptation to a program's execution environment; and (3) it explicitly separates performance, reliability, and machine dependent properties from type-dependent object functionality.

Using insights derived from experimentation with programs constructed with the CTK toolkit for multiprocessor systems, the thesis next presents a framework for building configurable parallel and distributed programs with modern object technologies. This framework supports a programming model where dealing with configuration issues is a central part of the design. A prototype for a distributed workstation platform demonstrates the framework's feasibility.

CHAPTER I

INTRODUCTION

There are many possible interpretations for what a **flexible system** is or which characteristics it should have. Efforts in achieving flexibility are usually described in terms of the specific bias of researchers developing such systems.

Software engineers may refer to a flexible system in terms of its basic architecture being well defined through reusable components and general integration mechanisms. Sometimes the term is used more generally, referring to broadly accepted goals such as portability, maintainability, and ease to making evolutionary changes. For the parallel programming world, flexibility in the design is usually a requirement because chosen levels of parallelism, scheduling, and synchronization mechanisms may vary based on the data and resources available. In the realm of operating systems research, flexibility means that different demands on the functionality supported by the operating system can be accommodated with reasonable performance, and that the services being provided will adapt to the execution environment in order to attain an efficient use of resources. The fact that distributed environments are becoming more common increases the relevance of issues such as heterogeneity and the complexity of usage, stressing the necessity for adaptations

that provide a balance between application needs, basic operational requirements in operating systems, and characteristics of execution environments.

Flexibility per se is not a new concept, as we can see its first manifestation through the early uses of self modifying code in operating systems. Since efforts suffered from a relative scarcity of computing cycles and memory, the production and management of complex flexible software system was unreasonably difficult, if not impossible. However, as hardware capabilities evolved, software technology has advanced and it became possible for program designers to consider a diversity of strategies and paradigms in order to match widely varying application requirements. Furthermore, the need for runtime flexibility has increased substantially due to the recent boom of new application categories, such as multi-media systems and the wide area distribution of information across the Internet. In all such applications, the attainment of reasonable levels of performance requires the exploration of specific characteristics of the execution environment and the execution of specialized behaviors for those.

Ideally, a flexible software element should be able to adjust itself to a current execution environment so that it mimics the performance of an object designed for that particular scenario. It is not clear whether this is a feasible goal even within the confines of a specific application, since it requires a complete understanding and parameterization of the execution environment and its relationship with software components.

Our work explores configurability issues in some specific situations and environments, the ultimate goal being the development of a framework for reasoning about and dealing with configuration issues. Specifically, we propose the construction of a framework for building flexible systems through the use of configurable objects. We aim to achieve performance increases by making use of the framework flexibility and configuration support, thereby addressing one principal requirement of current system technology. Many solutions for the problem of pursuing performance gains via adaptation are specific to some application domains, input data, or programming and executing environments. Combining performance requirements with framework generality is a major challenge. Our approach attempts to accomplish the framework's potential generality by focusing on two additional requirements: (1) we must address performance issues by considering the basic mechanisms that influence them; and (2) we must provide abstractions in order to incorporate flexibility in a system in a methodical fashion.

CHAPTER II

CTK: CONFIGURABLE OBJECT ABSTRACTIONS FOR MULTIPROCESSORS

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, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties.

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.

The work described in this chapter demonstrates the benefits attained from using policies and attributes, which jointly implement 'configuration abstractions' that have been separated from objects' basic functionalities. Such abstractions provide a convenient means for configuring objects during execution or for experimenting with alternative object implementations. Specifically, using policies and attributes, objects may be specialized using diverse techniques, including parameterization and interposition. Multiple specializations may be applied simultaneously by association of multiple policies with objects, resulting in dynamically configurable systems where attributes resemble 'knobs' being manipulated at runtime and policies implement the changes resulting from such manipulations.

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 thesis are the KSR and SGI machines.

2.1 Attaining High Performance by Dynamic Configuration

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, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties.

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.

This thesis demonstrates the benefits attained from using policies and attributes, which jointly implement 'configuration abstractions' that have been separated from

objects' basic functionalities. Such abstractions provide a convenient means for configuring objects during execution or for experimenting with alternative object implementations. Specifically, using policies and attributes, objects may be specialized using diverse techniques, including parameterization and interposition. Multiple specializations may be applied simultaneously by association of multiple policies with objects, resulting in dynamically configurable systems where attributes resemble 'knobs' being manipulated at runtime and policies implement the changes resulting from such manipulations.

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 thesis are the KSR and SGI machines.

2.1.1 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 component performance and (2) the runtime *configuration* of those aspects to maintain high performance despite variations in a component's use and in its execution environment. For example,

in complex multi-media and real-time applications, timing requirements and the components implementing the resource allocation mechanisms used to enforce these requirements are routinely separated from the applications' basic functionalities and adjusted at runtime[33, 42, 113].

Similarly, research in high performance programming and in operating systems has derived gains in performance and reliability from the dynamic configuration of multiprocessor synchronization constructs[15, 73], of network communications[49], of file systems[82], and of distributed object abstractions[18]. In all of these cases, basic component functionality is separated from the implementation attributes affecting performance.

Separation and configuration can also be the mechanisms for 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[45], by a location transparent invocation mechanism in the Lipto system[26], and by object migration in the Apertos system for uniprocessor platforms[112].

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 must often be performed dynamically (at runtime). 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[107].Similarly, runtime changes in lock implementations on multiprocessor programs can lead to substantial performance gains[73]. 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.

2.1.2 Contributions

The *Configuration Toolkit* (CTK) described and evaluated in this chapter is an object-based programming library that supports programmers in (1) separating object functionality from its performance or reliability attributes and (2) configuring objects on- or off-line:

- CTK permits the specification 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. For example, a policy may interpret runtime-supplied attribute values to vary an object's internal implementation without changing its functionality, or to vary the semantics and implementations of object invocations without affecting or altering the methods being invoked. Runtime

changes to attribute values are performed using additional 'attribute' parameters associated with object invocations or by invocation of methods defined on policy objects.

- 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 and to exploit peculiarities of the underlying multiprocessor hardware.

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

The general importance of this work is derived from two sources: (1) its implications on how parallel programs, library operating systems[31], or kernels may be structured, and (2) its exploit of configuration for object-based systems, which is particularly important for distributed object-oriented systems, as we discuss in Chapter 3. More specifically, this thesis' presentation of CTK and its use offers several interesting insights into the construction of high performance software. First, we describe the fashion in which basic object functionality may be separated from its configurable attributes, by designing a sample configurable multiprocessor object (see Section 2.4.1). Also demonstrated by this example is CTK's utility for implementing complex (user-driven) adaptive object behaviors, using only pre-defined classes and policies provided by the library. As a result, object configuration may be specified with simple language constructs and implemented without knowing

about the internal implementation of CTK's runtime support. Second, novel adaptive object behaviors implemented with policies and attributes are shown useful for attaining performance gains on multiprocessor platforms. Such gains are enabled by the low overheads of CTK's runtime support measured in Section 2.5.1. More importantly, performance improvements 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, including the replication of objects' configuration code and associated state. This demonstrates the importance of providing programmers with efficient mechanisms for implementing a variety of configuration techniques rather than offering a few predefined runtime adaptations. CTK's attributes and policies constitute such mechanisms. Third, policies and attributes may also be used to *extend* object functionality to utilize typical kernel-level operating system abstractions and services. Sample extensions described in Section 2.5.2 'convert' non-persistent to persistent objects transparently to end users, and they configure objects such that all of their invocations are monitored automatically.

2.2 Related Research

2.2.1 High Performance Computing and Operating Systems

Both the high performance computing and the operating system communities have used program configuration to improve performance or reliability. Historically, high performance applications have dealt with potential static or dynamic mismatches with the operating system by entirely removing it at runtime, as evident in early hypercube machines[36] that offer only basic runtime 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[51, 111], or to tailor programs to underlying hardware and systems using compiler support[94, 35]. We posit that such approaches should be combined with the configuration-based techniques presented in our work, thereby resulting in improved programmability, extensibility, and scalability of high performance applications.

A variety of research results has enabled the runtime configuration of operating systems in order to improve the performance of specific user programs, including the early work on policy-mechanism separation in Hydra[60, 22], the removal of operating system services from 'fixed' kernels to the configurable user level in the Mach[83] and NT[23] operating systems, and ultimately leading to the notion of

micro-kernels and user-level libraries for implementing customized operating system abstractions[16, 31]. In effect, such research has established the fact that application programs may be 'combined' with operating system functions such that both may be configured jointly, using the same programming techniques and software infrastructures – sometimes called 'operating software'[89]. Such joint configuration is explored in several recent object-based efforts[54], including the Choices[11], Spring[45], Chaos[87], Apertos[112], and ACE[85] operating systems. It is also being explored in efforts that address object fragmentation, including our own past research on hypercubes[91, 18], Shapiro's work on fragmented network objects[98], and recent work on object fragmentation by Tanenbaum[47].

Program or operating system configuration may be performed with diverse methods and at different times, including at compile-, link-, boot-time[34, 16, 31, 11]. Runtime configuration methods include the dynamic re-linking of OS kernels, server process creation and deletion[83], the re-direction of selected system calls using interposition in [52] and using subcontracts in Spring[45], the dynamic adjustment of meta-objects in [85], the dynamic adaptation of object and invocation implementations[74], the runtime adjustment of individual object parameters[88] or of structural properties of sets of communicating objects[1]. Compiler-based techniques for runtime configuration have included the dynamic synthesis of program code[70] and the re-direction of procedure calls[59].

Our research builds on and partly extends past work in configurable systems. Our

aim is not to explore specific configuration techniques, but instead, to attain improved performance in parallel programs by giving programmers user- or kernel-level libraries with which they may construct dynamically configurable software abstractions. Our group's earlier contribution to this goal was the construction of a parallel programming library that offers the well-known Cthreads programming interface[20] along with built-in self-configuring Cthreads abstractions (e.g., a configurable lock abstraction described in [73]) and with support functions for the efficient collection and, ultimately, display[43] of threads' runtime state. This thesis describes the next step in our group's research, which is embodied by the CTK object-based library layered on top of configurable threads and used for the construction of configurable objects on multiprocessor platforms. CTK and its *policy objects* were conceived and implemented[38] prior to Spring's *subcontracts*[45]. Moreover, policies generalize the notion of subcontracts since they also permit the runtime interpretation of additional invocation parameters (*i.e.,* attributes), which may be used for either controlling object behavior not related to object functionality (e.g., timing behavior) or for the dynamic adjustment of object characteristics determining such behavior (e.g., for performance).

2.2.2 Real-time and Object-based Systems

CTK's emphasis on the attainment of improved performance in parallel systems distinguishes our work from other efforts addressing distributed or real-time

systems[66, 95, 10, 57, 68, 105, 71], 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[9, 42], in which we investigate how the needs of real-time applications affect the object model and its required runtime support[39]. Similarly, recent work on the dynamic configuration of distributed and object-based systems[13] often concerns specific configuration methods or general (rather than high performance) frameworks for implementing dynamically configurable applications, whereas CTK is exploring flexible mechanisms with which efficient, configurable program and operating system abstractions may be implemented.

Recent work on reflective programming[65] partly performed concurrently with our research is now leading researchers to offer abstractions like 'meta objects', which are similar to the policy objects used in CTK[24]. 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 address configuration problems for which efficient runtime configuration is not crucial. For example, in Apertos[112], meta-objects are used to achieve object heterogeneity such that changes in object behavior persist despite *object migration* among different meta-object spaces. The resulting required runtime checking of meta-object compatibility is too computationally expensive for applications in which configuration may be highly dynamic or short-lived.

In SOM[24] or CLOS[78], reflection principles are used for changing object behavior by invoking the methods available to create and initialize classes, to compose their methods tables, etc. As a result, object descriptions may be altered at runtime by invoking the inherited methods for dealing 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 manipulation of policy objects that have been specifically created for purposes of object configuration. This makes CTK more suitable for attaining performance gains via configuration, whereas SOM and CLOS address issues like the evolution of compatible object libraries by configuration of entire object system structures.

2.3 The Configuration Toolkit (CTK)

2.3.1 Introduction and Example

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. Such basic functionality is enhanced with a number of built-in classes, attributes and policies, including no-thread, single thread, and multi-threaded objects, along with multiple invocation semantics, such as asynchronous vs. synchronous invocations, various invocation

state feedback, etc. In addition, programmers can easily implement object representations and invocation semantics tailored to their applications' specific requirements in functionality and performance, such as 'toggle' or periodic invocations for real-time applications and invocations resembling transactions that can use state information about other objects' invocations[41].

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

InvocationType: *enum* {synchr, asynchr}

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

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 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 (*i.e.,* 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 [42]. We have also experimented with a second extension of CTK supporting configurable communication protocols[61].

This thesis does not focus on the use of CTK for operating system configuration and extension. Instead, we demonstrate the flexibility, performance, and generality of CTK by constructing and evaluating a configurable abstraction in a specific parallel program, which is then extended with additional functionality similar to what is often provided by operating system services. Functionality extensions concern object persistence and the monitoring of object invocations. The specific configurable program abstraction is a global queue. As with other global abstractions in parallel programs (e.g., global sums, etc.), certain dynamic program characteristics (e.g., the natural orderings between queue elements due to their times of insertion into the queue) can be exploited in order to reduce the costs of executing certain policies associated with those abstractions (e.g., the policy maintaining some acceptable global ordering in the queue).

2.3.2 CTK Abstractions and Structure

CTK abstractions. In CTK, an application program consists of a number of independent objects which interact by invoking each other's operations (methods). Each object maintains its own state, and that state is not directly accessible to other objects. Objects' implementations can 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'`[1]. 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 execution thread for execution of the called operation, but also does not synchronize concurrent calls. A `'Monitor'` [46] is an object without execution threads that only allows a single call to be active at a time. It can also define *condition variables* on which calls can *wait*, thereby allowing other calls to proceed, until the condition variable is *signaled*. A `'Task'` (like Ada tasks) has a single execution thread. It defines a number of *entries* which can be called from other objects. All calls are performed in the context of the `'TASK'` and are taken one at a time. Typical CTK programs consist of complex objects constructed from the four built-in object classes. CTK can be extended by

[1]The built-in object classes in CTK are quite similar to concurrent object constructs offered in recent designs and implementations of object-oriented concurrent languages[14].

defining new policy classes and linking them to the kernel.

In order to separate configuration from an object's basic functionality, each object has two distinct views: (1) the *application view* and (2) the *configuration view*, the latter being roughly equivalent to recent notions of meta-objects implemented in systems like ACE [85]. 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 sample attribute for a class is one that describes a dynamically determined aspect of the internal representation of each of its instances, such as the 'number-of-spins' performed by a lock object before the caller is blocked.

- A policy can define the invocation semantics to an object by intercepting invocation requests and by defining and interpreting invocation time attributes that can be specified as part of the invocation. A sample invocation attribute is one that specifies dynamically determined limits on the permissible duration of an invocation for multi-media or real-time applications[42].

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

example, a policy could specify a new invocation mode, such as an *invoke_event_dependent*, defining it in such a way that the other objects could interact with the policy for determining how (or whether) such invocations should be executed. Such control is useful for protecting programs against excessive number of events during emergency conditions, for example. Other examples of extensions are discussed in Section 2.5.2 below.

Distinguishing application from configuration view is also useful for achieving interface uniformity when CTK programs implement operating system services or potential legacy applications. In such cases, existing application views remain, but configuration views may change. For example, CTK programs may attain high performance on different underlying hardware platforms by customizing the policies associated with certain program abstractions acting as vehicles for interactions of the program with the hardware or with lower levels of the multiprocessor kernel. Since policies are executed implicitly as a result of object creation and invocation, typical application programs do not perceive policy customization; they see only the objects and classes defined in their code. One exception to this rule is when an object explicitly invokes an operation of its own policy (referred to as a *policy interaction*), in order to exchange data and coordination information between the 'original' object and its policy. Such interactions are useful for program-driven dynamic changes to objects.

CTK structure and implementation. The structure and implementation of the

Configuration Toolkit are depicted in Figure 2.1 as consisting of three components: (1) configurable threads, which is the portable Cthreads package underlying CTK[92, 72], (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 was optimized.

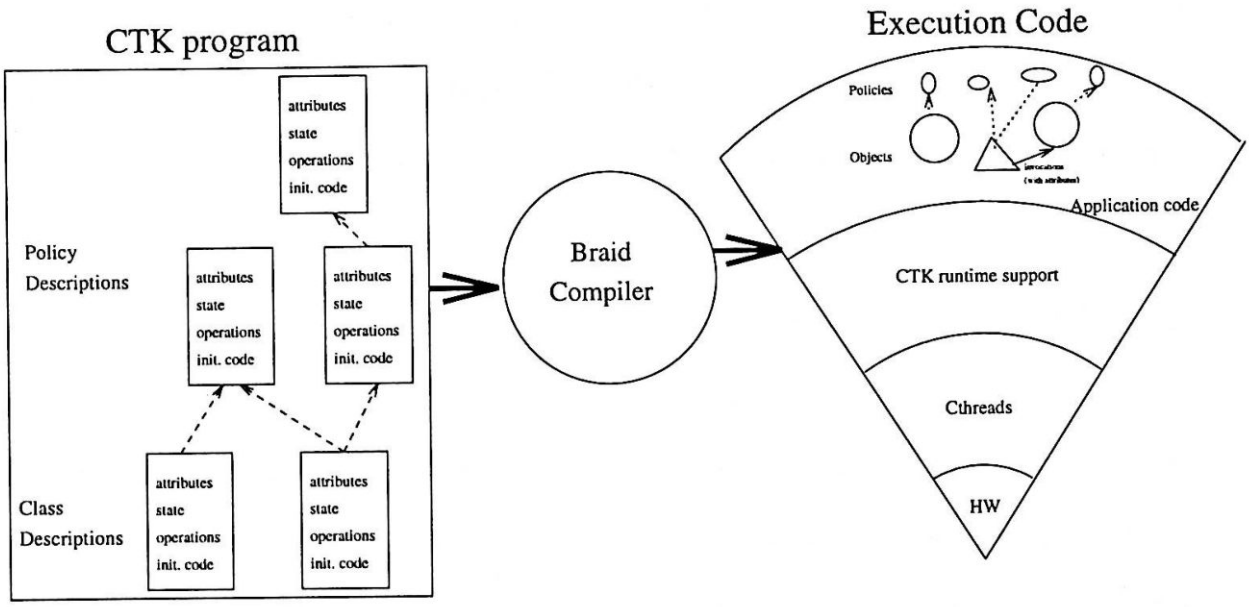

Figure 2.1: Structure of the Configuration Toolkit (CTK)

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. 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 Cthreads package) can be provided. Classes are specified using the experimental *Braid* 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. An example of a complex object is an internally parallel object containing multiple TADTs used as servers, with a policy that intercepts all invocations to the object. Another example explained in the next section is a complex object implementing a global queue object internally consisting of multiple ADTs serving as distinct object fragments.

Configurable Cthreads is the partially machine dependent component[72] 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.

2.4 Configurable Abstractions in CTK

2.4.1 A Configurable Distributed Queue

Motivation. This section demonstrates how *policies* and *attributes* may be used to configure the behavior of an object during its execution, using a sample complex object: a *distributed queue.* Many parallel programs and operating system services (e.g., schedulers) use such queues. The program studied in this chapter is a parallel branch-and-bound code, which uses a priority queue to store its subproblems that remain to be solved; the order in which subproblems are retrieved from the queue guides how the search space will be traversed. Clearly, the implementation of this queue can strongly affect overall program performance. Specifically, both a centralized implementation and one that maintains non-cooperating queue fragments local to each processor lead to poor performance, as demonstrated for several parallel platforms by our previous work. Extensive experimental results with alternative implementations of the Traveling Salesperson Problem (TSP) on distributed memory (an Intel iPSC machine[90]) and on shared memory machines (the BBN Butterfly and Kendall Square NUMA multiprocessors[17, 18]) have demonstrated that the dynamic configuration of selected attributes of the distributed queue is essential for good runtime performance. Specifically, 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[93, 17, 18], (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.

We next show that it is straightforward to implement an internally fragmented and configurable distributed queue abstraction with CTK's *policies* and *attributes*. In addition, by experimenting with alternative global element orderings and with the customization of queue access policies, substantive performance gains are realized for the TSP application. This demonstrates that CTK's configuration mechanisms provide appropriate functionality at acceptable costs.

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. More specifically, the aspects of *Queue* objects behavior chosen to be configurable in our example 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 (see [17] for implementations with a single fragment), 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[93], 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 its original measure 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.

```
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 2.2: Queue specification

Figure 2.2 depicts Braid-like pseudo-code for the queue abstraction. In this figure, the class *Queue* has the policy *DistributedQueuePolicy*. The state variable 'array_of_fragments' has an attribute describing the communication structure among fragments. This implementation permits each fragment to communicate with all other fragments, 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) for obtaining an item. The prototype for operation 'insert' specifies initial values for the attributes related to invocation mode, order of elements, and priority.

CTK objects can be created dynamically, or their creation may be specified at compile-time using the 'application description module'. In both cases, initial values may be specified for the attributes of each particular instance of a class:

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

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

The first line in Figure 2.2 specifies which policy class is associated with the class *Queue*: 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 are stored and manipulated by the policy object, and they will be used to dynamically control the execution of q's operations. In the example, the invocation 'invoke q $ insert(item)' will be intercepted by the policy object associated with q during creation time 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). 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.

```
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 2.3: The *DistributedQueuePolicy* class

The actual specification of the *DistributedQueuePolicy* in Figure 2.3 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 is must manage attribute values. In the current implementation of CTK, such additional operations use fixed naming conventions, and they address:

- *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]. The policy itself is an 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 this TADT 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. More specifically, an invocation request of the form:

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

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

is mapped to a call to an operation *invoke* of the policy *DistributedQueue-Policy*. 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 (.g., 'invoke_async' , 'invoke_sync', etc).

2.4.2 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, otherwise 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 (*i.e.*, the fragments) are perfectly distributed. The *DistributedQueuePolicy* shown above does not result in such complete distribution, as explained next.

Consider the situation depicted in Figure 2.4. Here, the distributed queue is mapped to three processors. Processors 1 and 2 host one of the queue fragments each; Processor 3 hosts one fragment and the queue abstraction (*i.e.*, the *Queue q* object) and its corresponding policy. The invocations labeled (1a) – 'insert' invoked by application code running on Processor 1 — , (2a) — 'remove' invoked on Processor 2 — , and (3a) — 'insert' invoked on Processor 3 — are performed concurrently. These invocations are 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 be referred to the appropriate 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!

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 provide 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

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

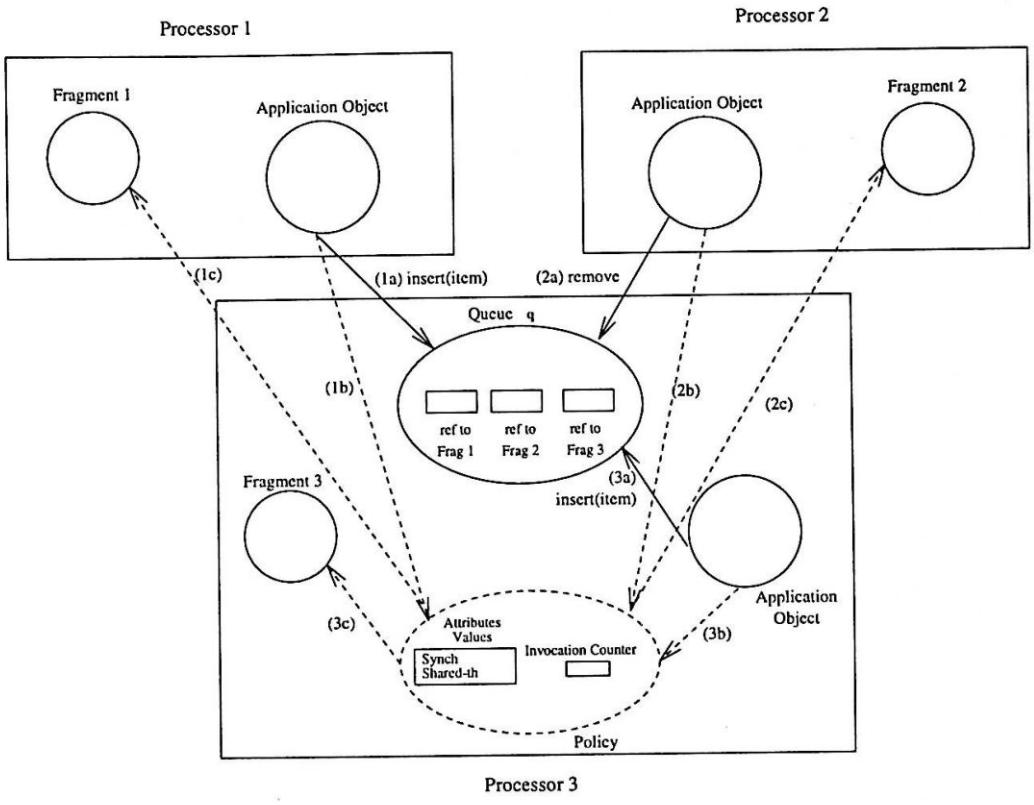

Figure 2.4: Invocation of a distributed queue using a single copy of a policy object

of the queue's basic functionality can be carried out locally. Figure 2.5 depicts the 'replicated' queue for the same situation as shown in Figure 2.4. 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 counts 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.

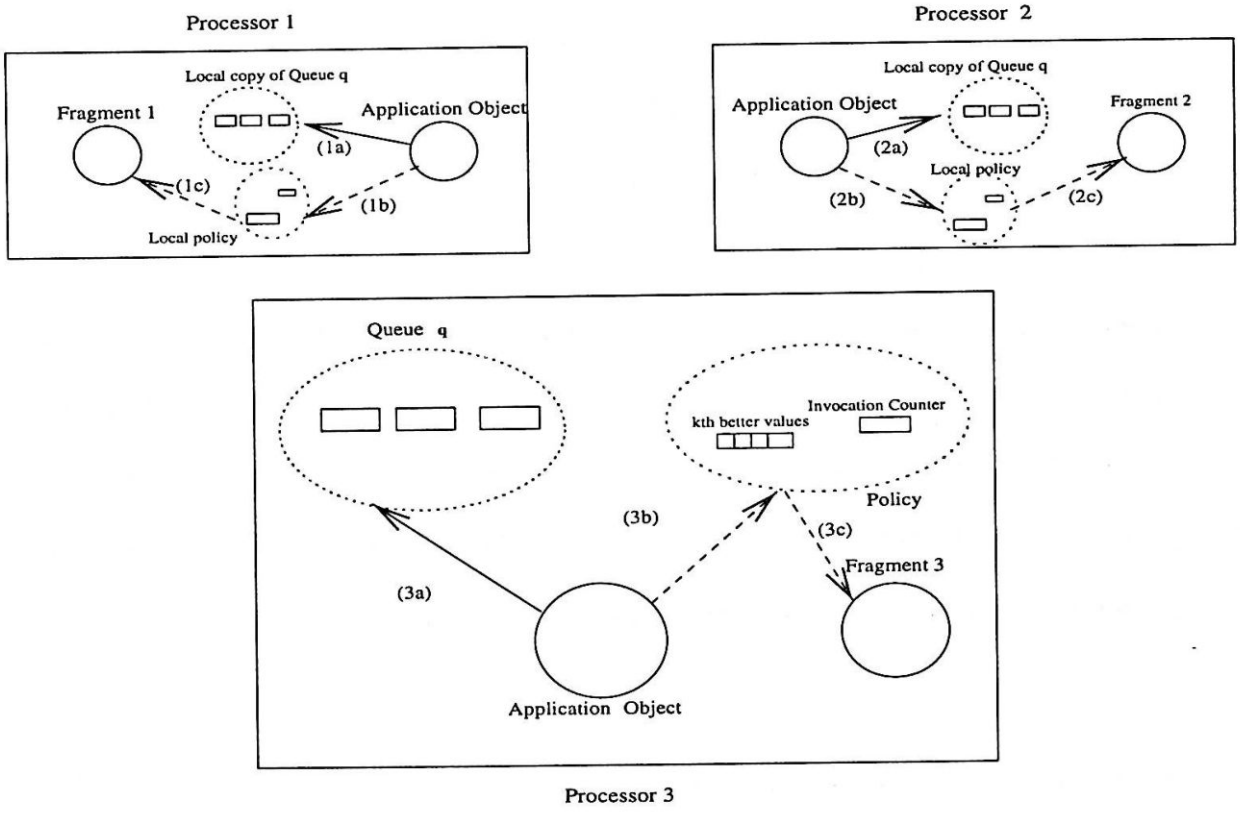

Figure 2.5: Invocations with local policy objects

2.5 Experimentation with CTK Objects

2.5.1 Performance of the Configurable Distributed Queue

Highlights. Two important results are described in this section. First, despite the apparent complexity of CTK's configuration support, its basic performance is sufficient for deriving significant performance gains from the dynamic 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 case, 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 showed 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. 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]. More

[4]The measurements in Table 2.1 utilize a non-optimized implementation of CTK for the SGI machine. For the TADT object type, the values in the table do not include forking costs.

importantly and as shown later in Table 2.8, more complex policies and additional invocation attributes can 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. The measurements in Table 2.1

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 2.1: Basic invocation costs in μseconds

are derived from runs on a 64-node KSR-2 supercomputer and on a the 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[42]. The KSR-2 machine is a NUMA (non-uniform memory access) shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes. Each node consists of a 64-bit processor, 32 Mbytes of main memory used as a local cache, a higher performance 0.5 Mbyte sub-cache, and a ring interface. CPU clock speed is 40 MHz, with peak performance of 40 Mflops per node. Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheap. 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 [72]. 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. However, all memory is uniformly accessible, so that the super-linear speedup sometimes attained with KSR applications does not occur for this machine.

Experiment description. 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 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. Specifically, 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 the collection of fragments implements the totally ordered (centralized) queue abstraction. The idea is to compute how closely each execution approximated a centralized queue's behavior on the average. For instance, if a distributed queue implementation actually returns the 'best' element for each removal (*i.e.,* 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 a uniform distribution 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.

Experiment outcomes. Experimental measurements demonstrate that a partial order scheme with a fragmented queue can implement a behavior close to that of a totally ordered centralized queue implementation. At the same time, the fragmented queue with its partial ordering has significant advantages in terms of queue execution times, even when queue accesses and element qualities are drawn from uniform distributions. Naturally, 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 2.2). However, 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%. The reason is that 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 2.2, 2.3, and 2.4. 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, 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 2.2 and 2.4 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 2.4. It is likely that they will also hold for other NUMA machines like DASH and multiple SGI machines coupled with high performance memory interconnections.

The synthetic workload tends to generate queues that remain small throughout the execution. 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 carried in the SGI (Table 2.4), "give-second-best" always offers better results than "get-second-best". In the KSR experiments, both protocols have similar execution times (as Table 2.4 illustrates).

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 2.2: Execution times for uniform distribution experiments (in secs.) (KSR-2)

Figures 2.6 and 2.7 depict the data from Tables 2.2 and 2.3 graphically. These

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 2.3: Quality measures for uniform distribution experiments (KSR-2)

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 2.4: Uniform distribution experiments (in secs.) (SGI)

graphs clearly distinguish the tradeoffs among alternative element distribution policies with respect to cost vs. ordering qualities. We also conclude from these result 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.

Experimental results with the TSP application using alternative queue implementations are depicted in Tables 2.5 and 2.6, and in Figure 2.8. 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,

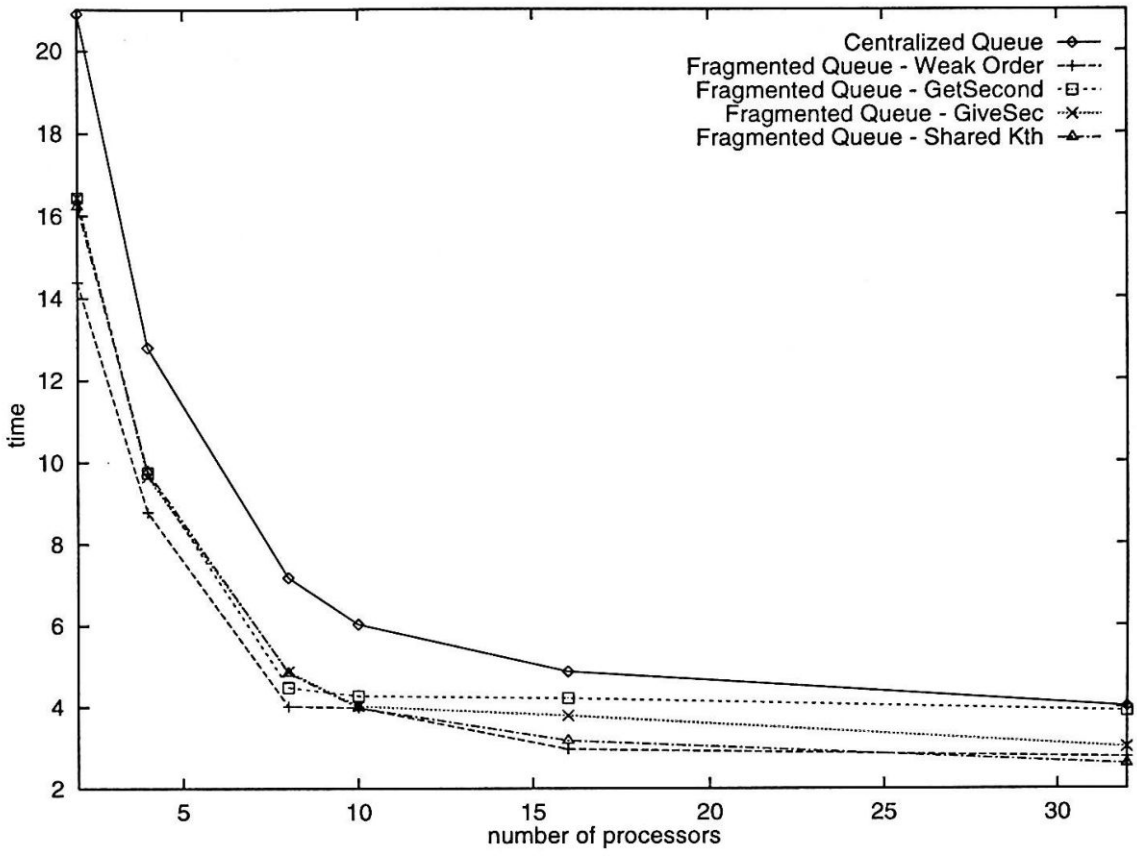

Figure 2.6: Execution times for uniform distribution experiments (in secs.) (KSR-2)

as specified in TSPLIB[5]. The graph in Figure 2.8 repeats the measurements shown in Table 2.5. It is interesting to note that the centralized queue outperforms the weakly ordered queue for smaller numbers of processors due to its superior queue quality, but with an increasing number of processors, the additional communication costs overshadow the quality gains. Then, once the number of processors gets larger than 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 central

[5]The Traveling salesman problem library(TSPLIB) provides researchers with a broad set of standard test problems[84].

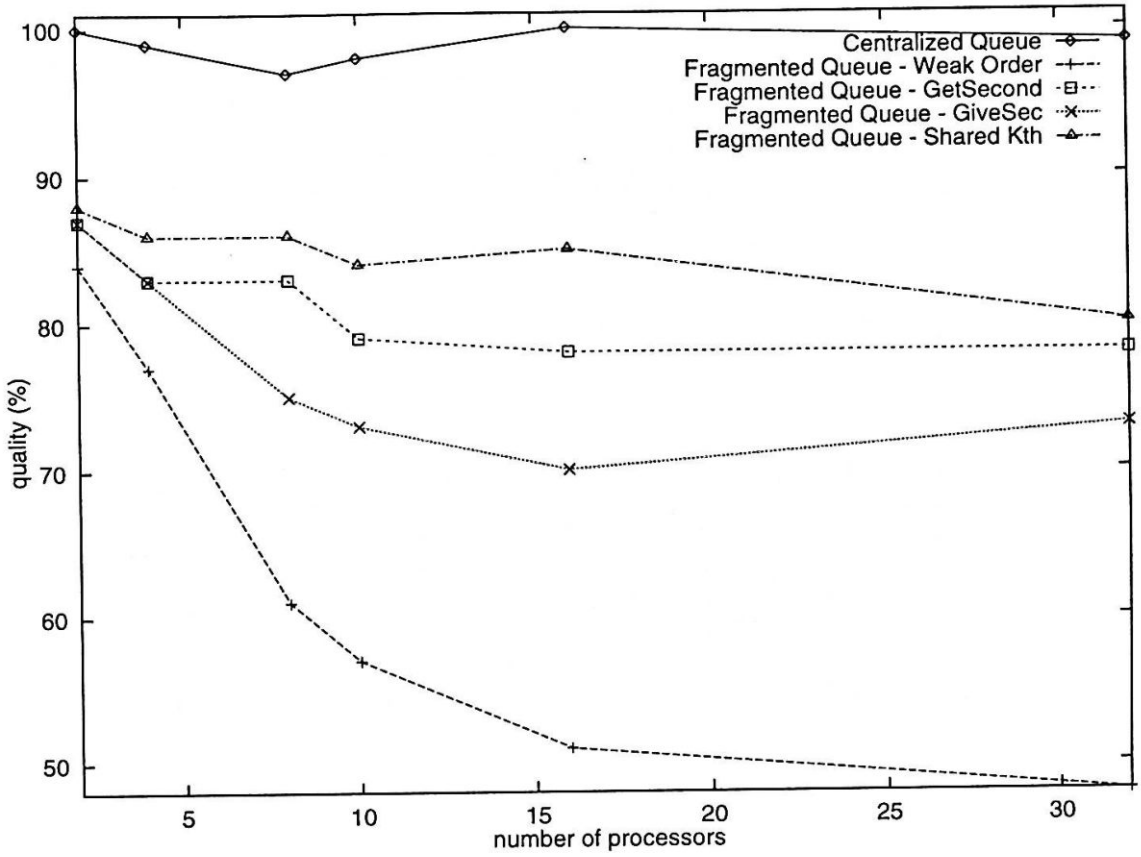

Figure 2.7: Quality measures for uniform distribution experiments (KSR-2)

queue TSP execution outperforms the weakly order again.

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 efficient and 'pay as needed' implementation of CTK's configuration mechanisms. More importantly, benefits from object configuration result from attributes and policies that can customize an abstraction's behavior to the needs of a particular application program, even during program execution. Since such policies' functionalities may be varied separately from basic

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 2.5: TSP execution times with configurable queue and for 51 cities (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	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 2.6: Queue quality measures for execution of TSP for 51 cities (KSR-2)

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.

2.5.2 Using Policies for Object Extension

Multiple policies: a monitored, persistent, distributed queue. Section 2.4.1 demonstrates the utility of CTK for performance improvement, by implementing and

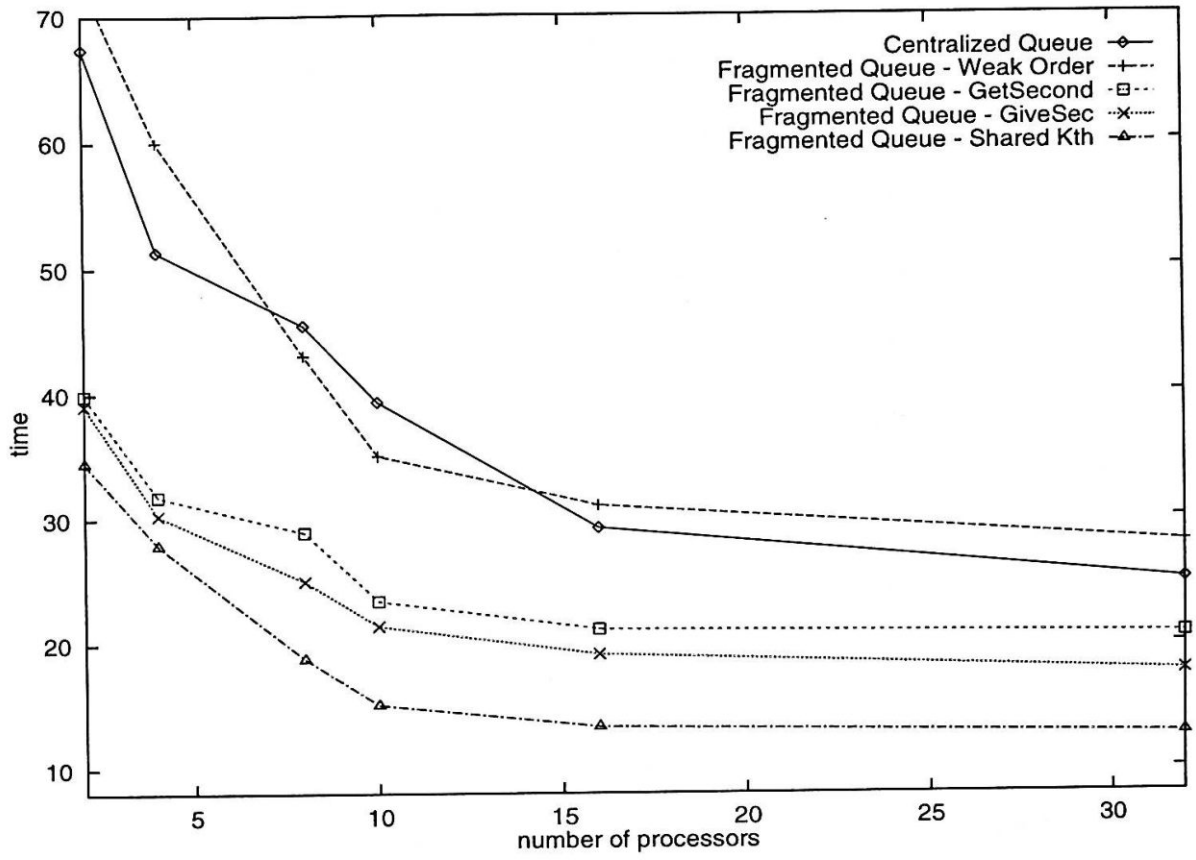

Figure 2.8: TSP execution times for 51 cities (in secs.) (KSR-2)

evaluating alternative policies and attributes for a distributed queue abstraction. More generally, such dynamic customization of abstractions — in order to adapt to the runtime application or 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

object	locality	nb of attributes	basic cost
Fragment (no policy)	local	0	15
Fragment (no policy)	remote	0	23
Weak Queue	local	1	33
Weak Queue	remote	1	36
Give-sec-best	local	2	33
Give-sec-best	remote	2	36
Get-sec-best	local	2	33
Get-sec-best	remote	2	36
Shared-kth	local	4	34
Shared-kth	remote	4	36
Shared-kth+Priority attr	local	6	38
Shared-kth+Priority attr	remote	6	40
total order	local	1	16
total order	remote	1	27

Table 2.7: Timing (μseconds) for invocations (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, 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/attribute 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 'stacked' policies.

Persistent objects. The implementation of object persistence described next does not attempt to explore alternative or efficient ways in offering persistence. 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.

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 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 [50] 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 for the TSP application 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 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 incorporate the identification of which queue fragment holds 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 another policy which internally utilizes the online monitoring mechanisms provided by the *Falcon* project [43, 96]. Using Falcon's basic mechanisms of sensors, local monitoring threads, and remote monitors, CTK's monitoring policy implements the following useful functionality:

- it time stamps 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.

2.5.3 Compiler Support for Optimizing the Use of Multiple Policies

Figure 2.9 shows the relationship between objects and policies for the monitored, persistent, distributed queue. With CTK, this complex abstraction is constructed

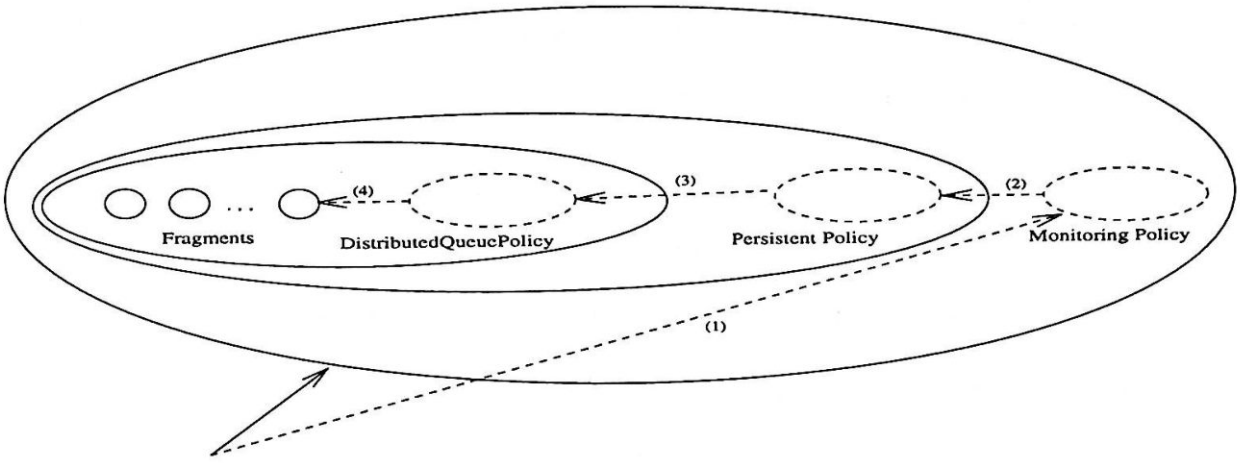

Figure 2.9: Monitored Persistent Distributed Queue object

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 2.9.

A straightforward implementation of multiple policies' association with an object simply permits each policy to interpret each object invocation using the appropriate attributes, then forward 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 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 changes the value of the attribute 'order' passed to the *DistributedQueuePolicy* to the value "weak", so that load balancing actions are not 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 2.8. 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 2.8: Invocation cost in μseconds for complex objects and policies

2.5.4 Discussion

It is unlikely that programmers can design solutions for how to configure objects like fragmented queues for dynamic applications such as the TSP code, because they would have to anticipate dynamic application's runtime behavior. This demonstrates the usefulness of employing CTK's lightweight configuration mechanisms for exploring which protocols for sharing abstractions result in performance gains. 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.

2.6 Conclusions and Future Research

The *Configuration Toolkit* (CTK) described and evaluated in this chapter 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 enables 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 permit the online configuration of object performance or reliability to match current application needs. As a result, CTK is shown useful for implementing the complex adaptive object behaviors required for high performance programs running on modern 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.

The work presented in this chapter demonstrated that performance improvements derived from the runtime 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 application programs, 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(Chapters 3 and 4, [2]), 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[40]: (1) in the application itself, (2) in the object transport substrate, and (3) at the protocol level. Our previous work has demonstrated the performance gains from configuration at each of these levels[49, 40], but we have not yet been able to show gains by simultaneous configuration at multiple levels of abstraction.

The extensions of object functionality described in this thesis 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 research progressed 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 chapter 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[44, 29, 43] The essential idea of this technology is to give users the ability to *steer* their programs quickly past uninteresting results or data domains, therefore significantly reducing program execution time or alternatively, offering additional computing power for required high-fidelity computations. We wish to understand the basic principles and opportunities of program steering, to develop abstractions and tools that facilitate the construction, execution, and control of steerable and configurable programs, and to demonstrate the performance advantages of program steering on parallel and distributed target machines. The object technologies presented in this thesis are an essential part of such efforts[28], in part because they permit such steering to be performed for any target parallel program, regardless of whether it implements specific applications or relevant operating system services.

The object technologies described in this chapter are now being realized for distributed systems as part of the COBS project, by generalizing the notion of policies to 'configuration objects', by describing the association of policies with objects as first class objects themselves, and by implementing object functionality using CORBA-compliant runtime libraries and compilation support. Namely, we are developing a framework within which object implementations can range from 'memory' objects permitting programs to share unstructured, raw data stored in

memory pages, to typed and structured objects explicitly defined by application programs. Moreover, the implementations of such objects can take advantage of underlying platforms offering multiple standard and custom communication protocols, and they can utilize the performance techniques required for their needs, including replication, caching, fragmentation, and dynamic adaptation of selected aspects of their implementations. Chapters 3 and 4 present the prototype we have developed for object configuration in distributed systems.

2.7 Lessons Learned

From this experience we learned that the separation between an object and its configuration policy is a convenient way to express and experiment with configuration. When methods are invoked, the policies associated with the object may interfere in the way functionality is achieved. We have shown that in CTK the interaction between object and policies can be carried within acceptable overheads per invocation. We have evidence, both in terms of performance gains and programmability, that this initial framework (objects + policies + attributes) models very well the flexibility concerns expressed by the parallel programming community in terms of varying level of parallelism, scheduling, and synchronization mechanisms. As previously mentioned, we used attributes and policies to express and change the number of threads associated with an abstraction and to vary the amount of synchronization being performed. Also, we were able to explore locality by using the CTK's basic

support for replicating objects and policies.

Other lessons concerned the relationship between an object and its policy. The initial design provided for a fixed and monolithic integration of object, attributes, and policies. These elements were specified through a specific language (the *Braid* language), and from these specifications C code was generated. For example, in the implementation of the DistributedQueue used in the TSP application the following aspects of the integration between the Queue object and the Distributed policy were fixed either by the Braid language or by the CTK design:

1. the association has to be done at compile time, *i.e.*, the class *Queue* is specified as having *Distributed* as its policy;

2. when the queue object is created, automatically a new object implementing the *Distributed* policy is created. This imposes a one-to-one relationship between objects and policies;

3. when one of the queue's methods is invoked, the invocation is automatically intercepted by the policy, more specially, by the method *invoke* which was required to be present in the interface of a policy[6].

The following scenarios exemplify the limitations implied by the three characteristics above:

[6]The mode of invocation (asynchronous, periodic, etc) determines which particular "invoke" (invoke_asynch, invoke_periodic, etc) will be used.

- it is not possible to use in the same application a *DistributedQueue* and a *DistributedPersistentQueue*;

- a queue can not "become persistent" during execution; instead it has to be created as a persistent queue with the persistence characteristics temporary disabled;

- in order to implement a *Distributed Persistent Monitored Queue* we want to abstract the simultaneous configuration actions being carried in different fronts (level of fragmentation, persistence, monitoring) by associating several policies to a same object;

- while trying to experiment with the number of threads serving a queue fragment, it became clear the necessity for a "N:N" relationship, meaning that many objects could be jointly managed by a policy. This enhances flexibility support by providing for situations where configuration should be driven by a global coordination involving many (basic) objects. CTK allowed the specification of objects as part of objects, but the object components were restricted to be defined by compile time, therefore were not useful to achieve "N:N" object-policy relationships.

In the TSP implementation we bypassed some of the limitations through the direct use of CTK internal data structures and routines. Some additions to CTK were done to better accommodate the application. For example, in order to model the situation where different policies are associated to the same object, we choose

to achieve an efficient solution by making the Braid compiler to generate one single policy object from the list of policy descriptions associated to the object in its class specification (*stacked policies mechanism*, as described in Document 2). The advantages are that we keep separation (the policy specifications are still provided through different modules) but manage all policy interference in the occasion of an invocation with the same cost of having one single policy object.

Notice that the stacked policies approach lacks generality: it assumes orthogonality between the configuration actions, therefore ignoring the possibility of "bad" (semantic) interaction between the different policies. The *stacking policies mechanism* was a first step towards supporting "1:N" relationships between objects and policies. "N:N" relationships were attained without high level support from Braid or CTK; we directly changed pointers in order to associate several objects to the same policy object. The schemes in figure 2.10 portray the several possible relationships between objects and policies: (a) is the semantics provided by CTK, (b) is the scenario where the *stacked policies* feature is used, and (c) exemplifies the situation where different objects are associated to a single policy (possible to be achieved in CTK through some pointer manipulation). (Figure 3.2 in next section exemplifies a general case not possible in CTK, but supported by our current framework).

There is one more aspect in which the CTK model for configurable objects itself does not seem to be flexible enough to be used as a general framework, as we can see through the *Distributed Persistent Monitored Queue* example. Building this configurable queue involves at the same time two sorts of configuration:

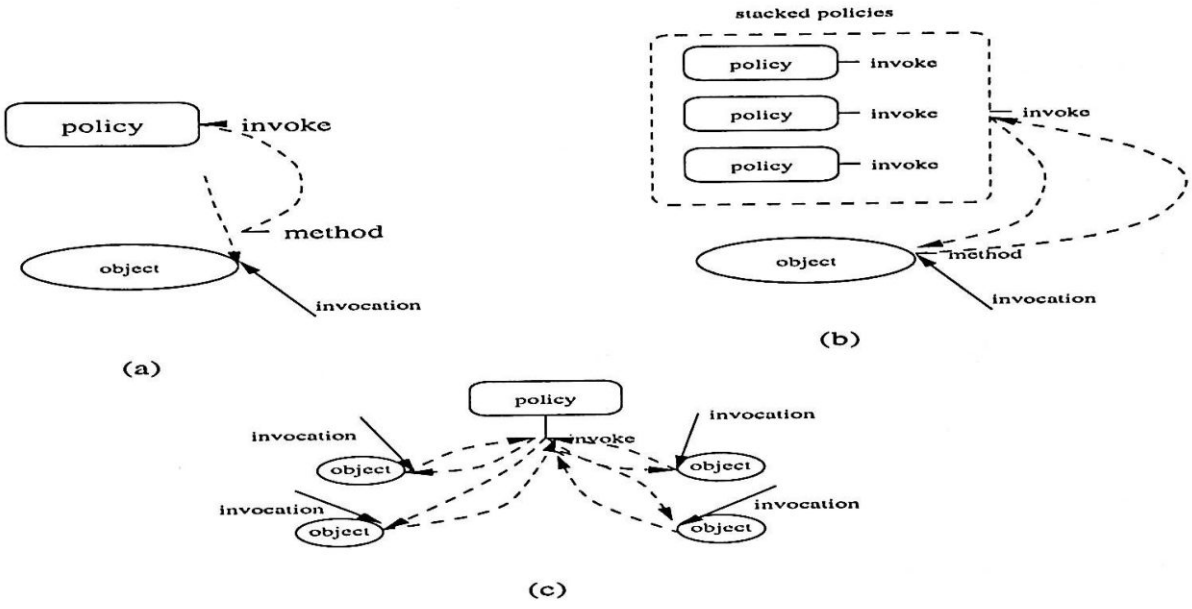

Figure 2.10: Relationships between policy and objects

- application level configuration policy related to the semantics to be associated to the abstraction. The Distributed policy deals with – among other things – the ordering among queue elements in the fragmented queue; the use of the abstraction in the application defines how flexible and meaningful varying the order is;

- operating system services (policies for "automatic" monitoring and persistence of the queue).

The example confirmed that separation among different configuration scenarios is crucial, and showed that in some cases the nature of configuration (*e.g.*, application versus operating systems related) requires a convenient way for integrating the policy to the object's methods execution. If the particular implementing strategy suitable

for the situation is not used in realizing the interaction between policy and object, the cost of achieving flexibility could become prohibitive or resources in the system could be unnecessarily overburden. In other words, the policy abstraction can be empowered by explicit specification of how objects and configuration policies should talk to each other. In general, application related configuration (dealing with the algorithm and computation being carried) will have more repercussion on object state and execution of invocations, and therefore it has its own needs in terms of how the policy and object can access each other state and activate actions back and forth. Configuration entities dealing with basic operating system services may require efficient execution to the point of making relevant saving procedure calls or memory as much as possible; these policies should be able to specify — by the use of attributes — specific "efficient" ways of uniting the policy work to the object behavior.

These considerations lead to the conclusion that the CTK runtime support interface and the Braid language constructs were not sufficient: the somewhat static interaction between object and policies offered by CTK decreases the degree in which flexibility can be pursued in the framework. Our current framework design (described in the next section) is being built on top of the implementation experience with CTK

CHAPTER III

A FRAMEWORK FOR DEVELOPING HIGH PERFORMANCE CONFIGURABLE OBJECTS

Software flexibility is as an important issue during the development of high performance and real-time applications, reliable systems, and in exploratory computing[104]. Furthermore, flexibility is perceived as a generally desirable software characteristic, since it facilitates the adaptation of a software product to new execution environments, usage constraints, and functionality requirements. The recent boom of new application categories, such as multi-media systems and the wide area distribution of information across the Internet, has led to further demands for flexibility in software. Namely, in all such applications, the attainment of reasonable levels of performance requires the exploitation of specific characteristics of their execution environments, typically by the execution of behaviors specialized for them. As a consequence, the software development process should address *runtime flexibility* as a crucial requirement for current and emerging application domains by incorporating adaptation capabilities into software components. Specifically, we aim to offer *configurable objects* as a means of achieving flexible systems in which runtime execution adjustments lead to improved performance. Namely, a flexible

software element should be able to adjust itself to its current execution environment in such a way that it can mimic the performance of an object customized for the environment. This not only requires the element to be configurable, but also capable of understanding its execution environment and its relationship with other software components.

Our work explores configurability issues. The goal of this work is the development of programming environment support for reasoning about and dealing with configuration issues. The framework we have constructed, COBS^{OM}, (1) addresses performance issues by considering the basic mechanisms that influence them; and (2) provides abstractions for incorporating flexibility into a distributed object program in a methodical fashion.

3.1 Related Work

Runtime flexibility per se is not a new concept, as evident from the early uses of self modifying code in operating systems. As hardware capabilities evolved, software technology has advanced and it has become possible for program designers to consider a diversity of strategies and paradigms in order to match widely varying application requirements.

Both the high performance computing and the operating system communities have used program configuration to improve performance or reliability. A variety of

research results has enabled the runtime configuration of operating systems in order to improve the performance of specific user programs, including the early work on the removal of operating system services from 'fixed' kernels to the configurable user level in the Mach[83] and NT[23] operating systems, the specialization of critical fragments of code in Synthesis[69], and the notion of micro-kernels and user-level libraries for implementing customized operating system abstractions[31]. In effect, such research has established the fact that application programs may be 'combined' with operating system functions such that both may be configured jointly, using the same programming techniques and software infrastructures. Such joint configuration is explored in several recent object-based efforts[54], including the Choices[11], Spring[45], Chaos[87], Apertos[112], and ACE[85] operating systems. It is also being explored in efforts that address object fragmentation, including our own past research on hypercubes machines[91, 18], Shapiro's work on fragmented network objects[98], and recent work on object fragmentation by Tanenbaum[47]. Efforts addressing distributed or real-time systems are [66, 57, 71, 9, 42].

For parallel programming, the chosen levels of parallelism, scheduling, and synchronization mechanisms may vary based on the data and resources available. In the work described in Chapter 2 and in [74], we achieved performance improvements through the dynamic adaptation of object and invocation implementations. We built a *Configuration Toolkit* (CTK)[102], which is a library for constructing configurable object-based abstractions implementing multiprocessor programs or operating system components. 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, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties. Using CTK, objects may be specialized using diverse techniques, including parameterization and interposition. Multiple specializations may be applied simultaneously by association of multiple policies with objects, resulting in dynamically configurable systems where attributes resemble 'knobs' being manipulated at runtime and policies implement the changes resulting from such manipulations.

The development of object oriented technology incited new ways of structuring implementations. It became common sense that in some arenas (*e.g.*, operating systems, real time systems, distributed applications) many implementation decisions did not represent "just details"; on the contrary, they had a crucial impact on performance ([54], [16], [99]). Object orientation was proposed as a way of promoting incremental design, robustness and incorporating diverse implementation alternatives. In particular, object oriented languages that directly support the use of *reflective programming* ([65]) and *meta-objects* ([37], [24]) such as Smalltalk and CLOS are advocated by researchers in the object oriented community as ideal environments for developing flexible systems. The *meta-objects* are similar to the configuration abstraction provided by COBS^{OM}. 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 address configuration problems for which efficient runtime configuration is not crucial. For example, in Apertos[112], meta-objects are used to achieve object heterogeneity such that changes in object behavior persist despite *object migration* among different meta-object spaces. The resulting required runtime checking of meta-object compatibility is too computationally expensive for our applications, in which configuration may be highly dynamic or short-lived.

As already discussed in Chapter 2, in SOM[24] or CLOS[78], reflection principles are used for changing object behavior by invoking the methods available to create and initialize classes, to compose their methods tables, etc. As a result, object descriptions may be altered at runtime by invoking the inherited methods for dealing with class objects, thereby attaining configuration by manipulation of a potentially complex class description. In comparison, object configuration in our work is relatively 'lightweight' since it involves only the manipulation of objects that have been specifically created for purposes of object configuration. This makes our framework more suitable for attaining performance gains via configuration, whereas SOM and CLOS address issues like the evolution of compatible object libraries by configuration of entire object system structures.

The *open implementation* approach was proposed in the context of meta-objects and reflective programming[32], and evolved into the use of those ideas in environments where performance requirements prohibit the maintenance of a complete

framework for interpreting and redefining object properties. In ACE[85]/ASX[86], the support for flexibility consists of the dynamic service-to-host mapping in a distributed system in order to effectively use available parallelism on multiprocessor platforms. Tigger [114] illustrates the use of *open implementation* to customize the behaviors of application-level objects in real-time environments.

David Shang proposed and implemented an object oriented framework where flexibility is attained by dynamic instantiation of parametric classes, focusing mainly on installation time configuration[97]; it differs from traditional work on parameterized classes or functions by offering dynamic instantiation and dynamic binding. Banerji and Cohn propose an infrastructure for building highly modular and flexible operating system components that can be composed to provide a variety of application specific system services[4, 5]). Recent work on the dynamic configuration of distributed and object-based systems[13] (such as Polylith[1], Regis[67], Equus[55]) often concerns specific configuration methods or general (rather than high performance) frameworks for implementing dynamically configurable applications. We describe all these approaches in more detail in Chapter 5.

3.2 Objects, Configuration Entities, and Configuration Channels

In this work, we assume flexible systems as being composed of abstractions that can be dynamically configured in terms of (1) their implementations, (2) how they use other resources in the system, and (3) their requirements in terms of performance, reliability, or application needs. Such abstractions can be built and tailored for specific needs by connecting a set of objects; some objects encapsulate the desired basic functionality while others carry out the work related to each one of the configurable facets of object behavior. In other words, our object abstractions:

- encapsulate some basic functionality;
- are able to accommodate dynamic changes in how their functionality is implemented;
- permit the dynamic addition or subtraction of features; and
- can express changes in execution behaviors and needs using attributes.

The intent of our novel framework for building such object-based abstractions is to:

- explore performance issues;
- offer mechanisms for achieving configuration that are lightweight and of general applicability;

- pursue flexibility simultaneously at many levels (ranging from user level objects to operating system services) in complex distributed applications;
- separate basic functionality from configuration issues, both being encapsulated in different components of the framework; and
- promote a model for designing flexible systems and reasoning about configuration possibilities.

The framework, COBS^{OM}, has three kinds of elements: (1)*objects*, (2) *configuration entities* and (3) *configuration channels*, which integrate (1) and (2) during execution. An *object* is described by an IDL interface[101] and an implementation module providing code for its methods. A *configuration entity* encapsulates the information needed to carry out actions related to configuring a given characteristic of an object. It is built separately from the object; the idea is that in the same way that we want to have classes of objects available when building applications, we also want to structure our flexible systems in a manner that classes of configuration may be reused. The application designer composes a configurable/flexible application element by coupling basic functionality (*objects*) to the components that describe each configuration aspect being explored (*configuration objects*). This approach makes "configuration" a first class element in our programming model. The usual object-oriented programming model, that comprises a collection of objects that communicate through method invocations, is now extended to include the presence of *configuration objects* that, once associated with an object, are able to direct the changes in its behavior. The association between *object* and *configuration object*

via *configuration channels* is explicitly and dynamically specified. The configuration channel provides information that determines how the interaction between the objects and configuration objects is implemented.

Before coming up with detailed mechanisms for the interaction (*i.e.*, how to make objects and configuration entities work together), we address a more elementary problem: *when a designer finds a convenient configuration entity in the configuration library, how can she assure that it can be integrated with the particular object to be enhanced with flexible behavior?* A complete answer to this question would imply the use of knowledge about the component's semantics. In general, applications where high performance is important do not employ programming environments, languages or tools that make information about the software elements at such a level of detail. Research results in analyzing class hierarchies for reuse[3] indicate that, even with appropriate formal models and specification levels, the solution can be too computationally intensive to be used at runtime. In COBS^{OM}, we adopt a simple solution for checking if objects and configuration entities can be integrated into a configurable software element. Namely, we define *compatibility* in terms of the the basic object's interface and the information available in the configuration entity's description. The configuration entity specifies its requirements on the object by enumerating the methods it expects to have available in the object's interface. We refer to these methods as *required methods*; they represent hooks that can be used by the configuration object in order to (1) get information from the object and (2) impose behavior or state changes that may be needed so that configuration actions can be carried out.

Figure 3.1 pictures three basic objects (*PriorityQueue, SimpleDataBase,* and *LinkedList* and one configuration entity enumerating its required methods. Objects *PriorityQueue* and *SimpleDataBase* are both compatible with the (partially depicted) configuration entity; object *LinkedList* is not.

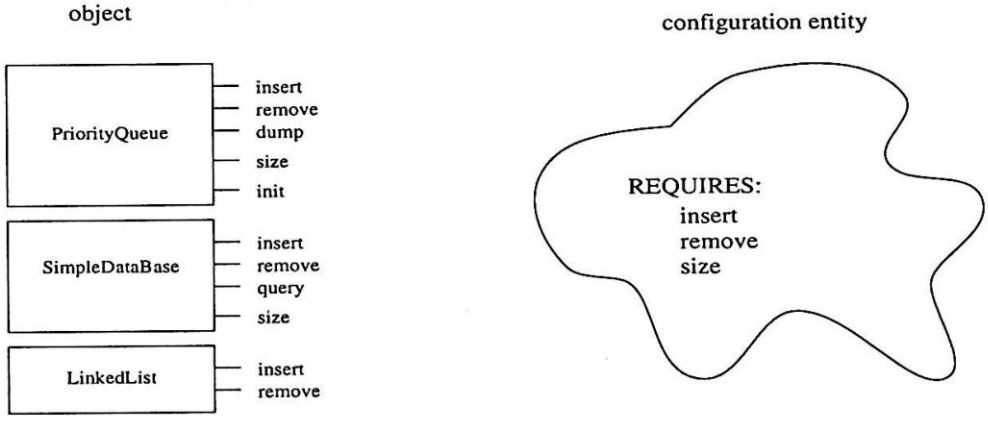

Figure 3.1: Exemplifying the compatibility notion between object interfaces and configuration entities

Notice that the design and implementation of a configuration entity module should not rely on any specific information of the object being configured, since at compile time, the only information about such an object is that is provides the *required methods.*

Multiple objects from different classes may be simultaneously attached to a given configuration entity, thereby allowing a single configuration object to manage the configuration of multiple basic objects. When the configuration object invokes one of the *required methods*, the runtime system has to map this invocation into the

respective method belonging to the specific object that is interacting with the configuration object when the invocation is issued.

Configuration entities are objects, therefore they also offer an interface. The methods in this interface represent configuration actions that can be initiated by explicit application demand. In this sense, the configuration entity is expanding the basic object's interface by offering configuration-specific methods.

The configurable objects composed by the association of objects and configuration objects can be varied at runtime, with parts being *efficiently* added or eliminated dynamically. More importantly, this association can be specified at the operation level, allowing a single object to carry out very different configuration approaches, accordingly to which method is being invoked.

Configuration channels abstract how invocations on the object interact with the configuration entity. They represent the link integrating objects and configuration entities, and they define how implicit configuration actions are activated during execution. Configuration channels are defined in the specification of the configuration entity, *i.e.,* for each available channel to a configuration entity is provided information about its identification, the code to run once the channel is activated, and characteristics determining how the channel abstraction is realized in an implementation.

Figure 3.2 portrays one object, three configuration entities, and configuration channels. This example shows how configuration entities and configuration channels can be used to compose a flexible distributed, persistent, monitored queue. In our previous work we showed that employing such a configurable queue in the implementation of a branch and bound algorithm for the Traveling Salesperson Problem can result in significant performance gains[102]. The queue behavior can be dynamically changed by removing and inserting configuration channels, and by changing the values of attributes that specify how a configuration entity acts. For example, by eliminating the link between one operation in the queue and the *Monitoring* configuration entity, selective monitoring can be achieved. By varying attributes values of the *FragmentedList* object, we can change how queue distribution is carried out.

When a configuration object is associated with an object (in our interface, via the *binding* call), it becomes possible to the configuration object interfere with the behavior of the basic object. Channels abstract association between entry points in the object (methods) and configuration (channel specification in the configuration entity).

Channels can be opened in two ways:

- automatically, when a configuration entity is bound to an object, either at compile time or during execution. This may be useful for composing applications where few configuration channels are available, and there is an "all-to-all" relationship among objects and channels (Figure 3.2); and

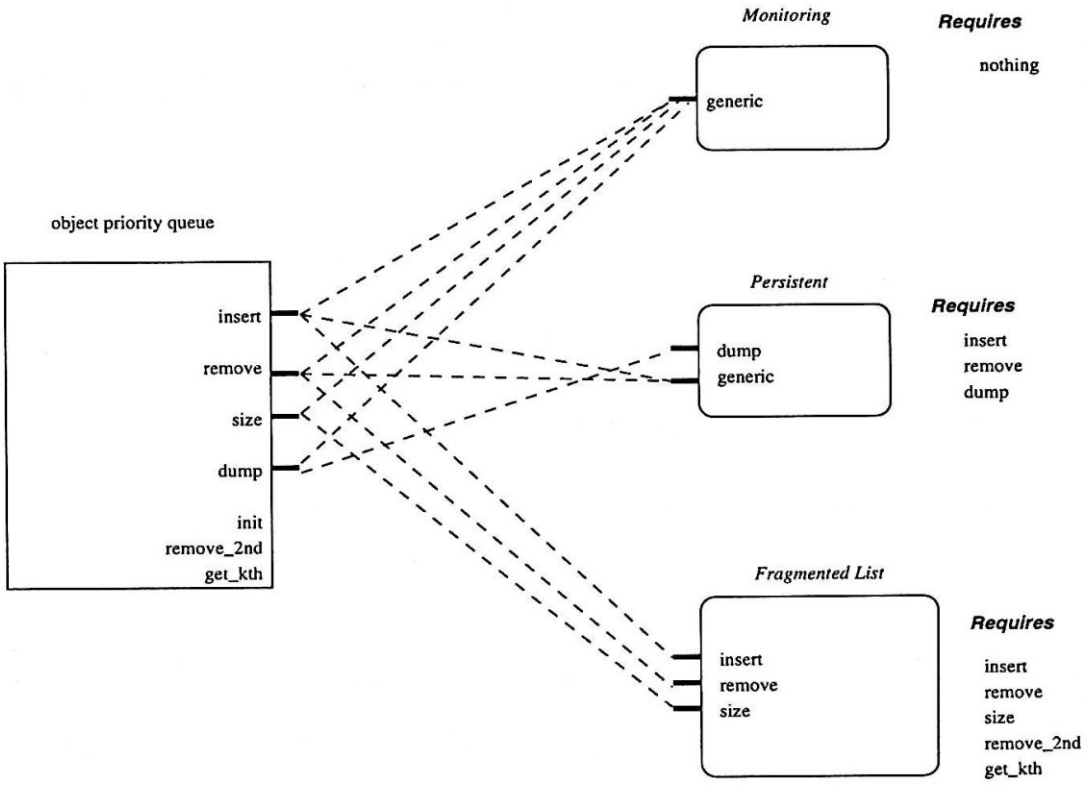

Figure 3.2: Configuration channels associate objects with configuration entities

- by explicit instruction from the application, which specify which method should be associated to which channel.

Parameters that determine channel's implementation behavior include: (1) attribute values in the configuration entity description, (2) values provided by the object when it initiates an *open_channel* operation, and (3) the runtime library default values. All such attribute values can be overridden at any time, thereby changing the channel's behavior. The channel attributes available in COBS^{OM} deal with the following issues:

- reuse of parameter blocks;
- how to send information (operation parameters, attributes) through the configuration channel:
  - through a function call;
  - forking a thread for running the channel code;
  - through a condition signal waking up a pre-existing thread; and
  - by explicit object invocation (through the *Object Transport Layer* developed by our group);
- an indication of whether the object should wait for the execution in the configuration entity's "side" of the channel;
- an indication of the method to be executed after the configuration entity's actuation; and
- an indication of some extra acknowledgment being sent by the object through the configuration channel to the configuration policy, after the channel code returns.

The object performs the following processing when responding to the invocation of a method currently bound to a configuration channel:

- identify which invocation attributes relate to each active channel;

- deal with parameter block creation (and marshaling, if needed);
- send the information through the channel;
- carry out post-configuration actions, as specified by the channel attributes.

The configuration primitives offered by COBS^{OM} were designed to offer complex configuration support efficiently. In the example in Figure 3.2, the "fragmented list" configuration entity turns a simple "priority queue" object into a distributed one: it generates name server information to be given to the transport layer, it creates the "priority queue" fragments on the multiple nodes, it allocates one local configuration entity to each of the nodes, and it manages distribution transparently to the queue object's user. COBS^{OM}'s support facilitates the development of such a configurable abstraction, but the composed element would not be useful if the overheads imposed by configuration interactions were high.

Configuration entities are implemented through objects, and therefore they can also be configured by association with other configuration entities, resulting in complex hierarchies of objects and configuration objects. Figure 3.3 illustrates a hierarchy composed by three kinds of basic objects: (1) *BTree-A* and *BTree-B* are instances of a binary tree data structure, (2) *Database* is a complex object that has a collection of *BTrees* as part of its state, and (3) *Device-1* and *Device-2* represent output devices. The other objects in the figure, represented by ellipses, are configuration objects. *PersistenceConfig* is adding persistence to the behavior of *BTree-A*,

BTree-B, and one of the *Database*'s internal binary trees. *PersistenceConfig* itself is linked via channels to configuration objects *CheckPoint* and *IncrementalLog*. As a result, the way in that persistence actions are carried out can be changed dynamically. *Device-1* is a component of *PersistenceConfig* that can be associated with *SocketOutput* or *FileOutput*. These two configuration objects manage multiple *Device* instances by storing one file descriptor (*fd*) for each basic object they configure.

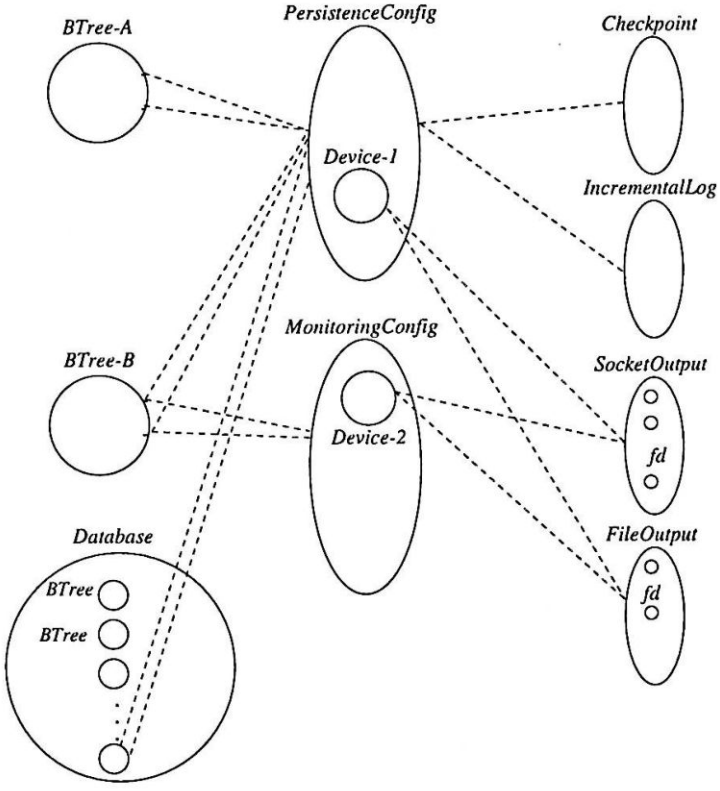

Figure 3.3: Hierarchy of objects and configuration objects

In summary, COBS^{OM} offers the approaches for flexibility considered more important in the literature:

- *Parametric variation*: the application ensures that relevant information about current demands and preferences about an available service are transferred to the service, which will then take appropriate actions to meet their requirements. This appears in the COBS^{OM} through the use of *attributes* that are passed from invocations to the configuration objects. Each configuration object defines and enforces semantics to its attribute values.

- *Interposition*: the flexible service maintains a fixed interface, but application-level code can be interposed between the uses of the service and the available service implementation. In this fashion, the application developer incrementally changes the semantics of a service, without altering the service itself. In COBS^{OM}, this can be easily achieved by dynamic association of configuration objects with specific methods (services) via configuration channels that are created using the default values for channel attributes.

- *Synthesis*: the application developer is allowed to synthesize an additional service, by specifying both its interface and its implementation. These new services should be treated as basic services, for example as an extension to the operating system or basic parallel programming support. An example of this is the synthesis of a distributed persistent monitored queue at runtime.

3.3 Conclusion

The framework presented in this chapter provides a programming model and environment where flexible software can be developed by designing configurable objects. COBS^{OM}'s 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, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties. *Objects* and *configuration objects* can be combined in complex ways, and the composition can be changed dynamically without the imposition of unreasonable overheads. Our experience in building configurable objects, in particular the Data_Object presented in Section 4.4, indicates that COBS^{OM} is suitable for building high performance distributed and parallel objects that can achieve flexible and complex behavior in runtime. Moreover, the programming model offered by COBS^{OM} encourages incremental design and reuse of components.

CHAPTER IV

A PROTOTYPE FOR DISTRIBUTED PLATFORMS

This chapter describes our framework for the construction of configurable objects in distributed environments: COBS^{OM}. This prototype offers the programming model described in Chapter 3 through a library that supports objects, configuration entities, and configuration channels. Tools are available for the composition of configurable objects from IDL[77] specifications and configuration descriptions.

Section 4.1 and 4.2 describe the prototype we built. Section 4.3 describes an object built with COBS^{OM} that is used to store information about object references and facilitate the development of fragmented objects. Section 4.4 introduces the Data_Object, a complex configurable object built with COBS^{OM}.

4.1 Design Issues

COBS^{OM}'s design incorporates insights derived from our previous work on supporting configuration in multiprocessor environments[74, 102]. Our experience with CTK (Chapter 2) demonstrates that (1) complex configurable abstractions require *n-to-n*

relationships between objects and configuration components in order to achieve efficient use of resources and (2) support for object replication facilitates exploitation of locality and permits efficient implementation of configurable objects in distributed memory environments.

As described in Chapter 2, CTK toolkit provides compiler support for a C-extension programming language that combines object implementation and configuration component descriptions into a single module. Besides controlling initialization and usage of the underlying thread subsystem, the CTK's runtime system entails detailed information about classes, objects, operation's signatures, and compile-time configuration setup. The design requirements for COBS^{OM} reveal a different direction:

- In order to facilitate usage and integration with other applications, new languages constructs were considered undesirable; and
- The amount of static information about objects manipulated during runtime should be kept to a minimum, thereby allowing efficient management of distributed objects.

COBS^{OM} takes a library approach, by this means increasing the framework strength in three ways: (1) legacy objects and implementations can be integrated with COBS^{OM} objects, (2) configuration appears as a first class element in the model, *i.e.*, , application programmers are able to manipulate configuration abstractions and freely alter their composition, and (3) parts of COBS^{OM}'s functionality (*e.g.*,

the support for object fragmentation) can be easily reused.

4.2 Implementation Issues

Objects are described by an IDL interface. The object developer provides an implementation module which associates code with the methods present in the interface. Using an IDL compiler front end available from the OMG site[76], we have constructed an IDL to C compiler. This tool parses IDL specifications and generates (1) a header file describing defined types and method prototypes and (2) a meta-description of the interface. This intermediate description is introduced so that the IDL to C compiler front end is decoupled from our object model support and so that the back end module is easily changed as COBS^{OM} evolves. Appendix B defines the meta-description format.

The tool code_gen consumes the meta-description of an IDL interface and generates class-specific implementation routines for the creation of objects, parameter block allocation, and method invocation. For each IDL complex type (sequence, array, or structure) present in the interface, code_gen generates routines for marshaling/unmarshaling arguments. It also extends the usual IDL attribute semantics by providing a mapping[1] to the attribute implementation described in [2]. As a result, attribute manipulations can be carried out efficiently, and attribute values

[1] In the current COBS^{OM} implementation, the mapping is available for the following IDL types: *string*, *enum*, *boolean*, *char*, *long int*, *short int*, *float*, and *octet*.

can be uniformly retrieved from and propagated to lower application levels and subsystems. These attributes are also employed in the implementation of IDL *contexts*. Moreover, code_gen exports the interface description to the *Interface Repository*, so that available class information can be queried at runtime and composition compatibility can be checked. The PBIO library[27] is used to achieve inter-operability among descriptions generated by heterogeneous hardware.

The configuration entity descriptions are processed by the tool conf_gen. As with code_gen, code is generated from the description to deal with the manipulation of configuration entities. Also, a configuration description meta-information file is produced and stored in the repository.

In order to facilitate the process of building applications, we allow their specification in terms of which classes (of objects and configuration objects) they use. From this description, the tool app_gen generates a makefile, ensuring that all necessary compilation steps and COBS^{OM} tools are applied.

Figure 4.1 pictures how the tools are used for building an application that uses a queue abstraction and a configuration entity that provides replication.

The formats for describing configuration entities and applications (input data to conf_gen and app_gen, respectively) are presented in Appendix B.

The framework library provides runtime support for objects, configuration objects, configuration channels, and manipulation of basic IDL types. COBS^{OM} can be used with a threads package, thereby allowing the use of its concurrency/parallelism

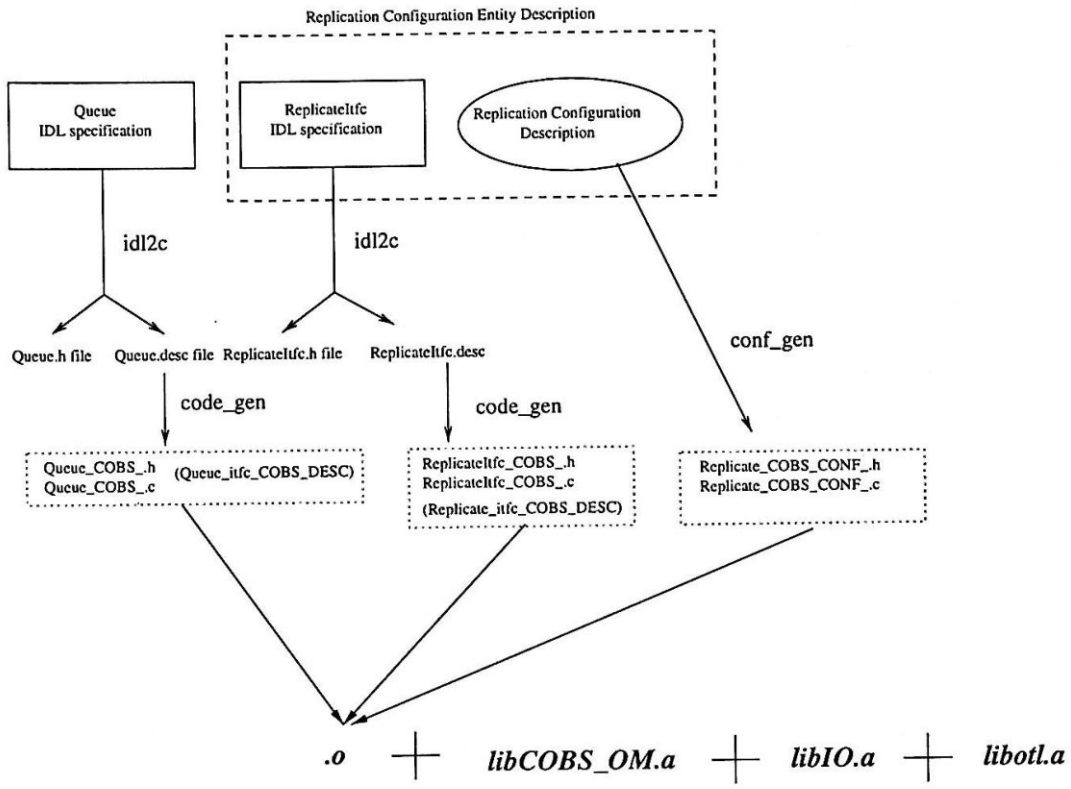

Figure 4.1: building an application in COBS^{OM}

mechanisms in the development of COBS^{OM} objects. The current implementation has been tested with Cthreads[92] and Solaris 5.5 Pthreads.

Another relevant function carried out by the library, code_gen, conf_gen, and app_gen is the activation of initialization steps when distributed applications are being built up. The current COBS^{OM} implementation has been integrated with the *Object Transport Layer* (COBS^{OTL}) package. COBS^{OTL} offers support for efficient and flexible remote invocation, so that COBS^{OM} objects can be distributed across networked nodes. By adding an extra call and specifying a few attributes, applications designed and tested with only local objects become distributed. The interfaces

to these objects remain the same, regardless of the nodes on which the COBS^{OM} objects reside.

The current implementation relies on COBS^{OTL} for management of object naming, efficient identification of local object references, and propagation of server information (host name, port number) among the nodes participating in the application. The tool code_gen generates dispatchers, object creation attributes, and remote invocation information in the format expected by COBS^{OTL}.

Appendix A presents the COBS^{OM} interface.

4.3 The Object_Server and Support for Fragmented Objects

The Object_Server is an object built with COBS^{OM} that can be invoked from any application. It implements a database for keeping information about COBS^{OM} objects. By querying the Object_Server, applications can retrieve name information about the objects they intend to use. Global, application-specific, and host-specific searches are available. The Object_Server is a multi-threaded object. Its methods can be executed synchronously or asynchronously.

The Object_Server functionality includes basic support for fragmented objects. In COBS^{OM}, it is possible to create complex objects whose components are distributed

across multiple nodes. Invocations of a fragmented object can be automatically delegated to a local fragment, if present. Applications can register fragments in the database and retrieve information about fragment distribution. The Object_Server interface includes a method for inserting fragments into the database that will block the invocation until all participating fragments are available. This facilitates the developer's task by providing implicit synchronization of composite object creation.

The Object_Server has been very useful in the implementation of Data_Object, a configurable object described in the next section.

4.4 The Data_Object Abstraction

In this section we describe one configurable abstraction built with COBS^{OM}: the Data_Object.

Runtime adaptation of high performance scientific applications has been investigated by our group for many years[29, 28]. In collaboration with atmospheric scientists at Georgia Tech, we have developed a parallel and distributed global chemical transport model[56] capable of running on any of the high performance engines in our computing environment (Figure 4.2). Models like this are important tools for answering scientific questions concerning the distribution of chemical species such as chlrofluorocarbons, hydrochlorofluorocarbon, and ozone. This model generates a very large set of data at each time step of the simulation. As a result, making

the data available to the end user brings out problems similar to data mining in large database systems. Using COBS^{OM}, we have developed a configurable object that offers flexible and efficient access to the model's data. The Data_Object makes the output data from the Atmospheric Application available to the tools performing visualization and application steering[28].

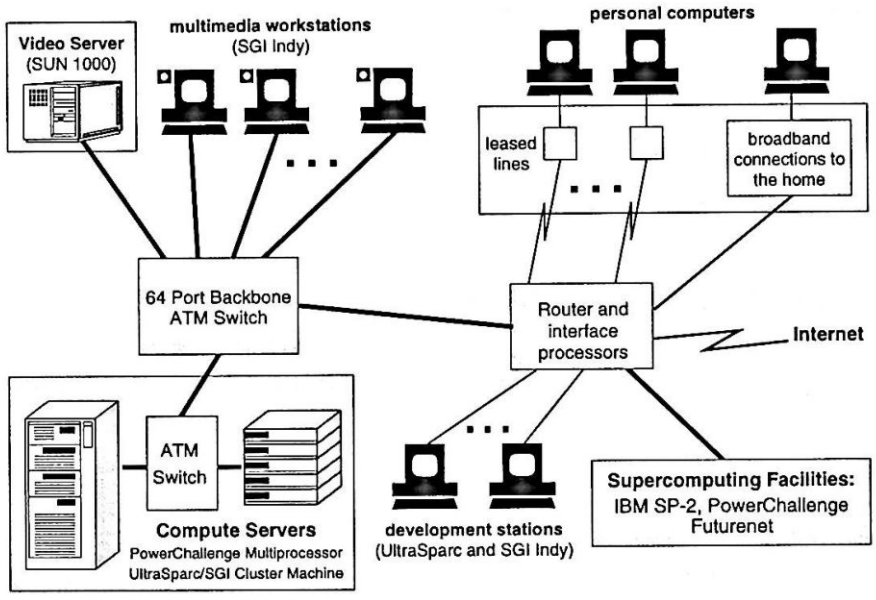

Figure 4.2: The computing environment in the Distributed Laboratories Project at Georgia Tech.

Like most scientific applications, the global chemical transport model produces a large amount of data. Through the Data_Object's uniform interface, interactive tools can access this data flexibly:

- Data can be obtained from a running model or from stored results of previous executions. In both cases, the data source may be distributed across several computing/storing nodes.

- Data_Object's users can request the specific simulation time steps and atmospheric levels in which they are interested, thereby decreasing the communication costs involved in attaining data at the granularity produced by the model (all levels for each time step).

- The Data_Object can deliver data to the visualization tools in both spectral and grid domains. Even though using the spectral format is a compact representations of each atmospheric level's data, for small grid regions, communication costs can be decreased by having the spectral-to-grid point transformations performed at the data source's nodes. Moreover, this transformation is time-consuming and benefits from its execution on a parallel platform[56].

- multiple users can simultaneously examine the data, by simply invoking the Data_Object's interface.

Data_Object is a convenient means of accessing application data for several reasons:

1. It offers a natural interface for requesting specific sets of data. For example, for the interactive steering support tool developed by Heiner and Zou[28], the Data_Object method

   get_grid (*latitude/longitude range, range of levels, range of time steps*)

   is sufficiently general to address all the existing tool requests for application data. Moreover, the level of abstraction in the method signature matches the

tool's view of data.

Also, data is still available in the spectral format. The 'data examiner' is able to specify filters or watch for interesting values in either domain.

2. Data access tools may be oblivious to data source implementation details. The Data_Object's encapsulation of model's data simplifies tool development tasks by hiding the issues related to data supplies. For example, in the current implementation data may be either stored in files or available from a running application as a stream of events. Other approaches for obtaining model's data can be added without changing the tool. Similarly, the current implementation supports distribution of data across computing/storing nodes. The distribution is flexible and transparent to the object's users. As a result, exploration of techniques for minimizing communication and access costs (*e.g.*, data caching) can be carried out independently of the tool's development.

3. Conversion of data among representation domains is usually application dependent. Factoring this conversion out of the accessing tool increases modularity and facilitates reuse.

Figures 4.3 and 4.4 show a high level picture of the Data_Object as a composition of multiple object fragments. These figures will be refined in the next section, where the Data_Object's design is described in more detail.

Figure 4.3 portrays the Data_Object layout when data is retrieved from two files produced by the model. The object is fragmented such that the interface to the

application code (*e.g.,* visualization tool) is available in fragment A. When a request for grid point data is issued, A will infer which node(s) store the data and then invoke the appropriate(s) object fragment(s). The application can dynamically attach filters to object fragments, thereby refining the data before it is sent through the communication link. In Figure 4.4, data is retrieved from a running model through sockets. The Data_Object initiates its execution by opening connections to the sockets through which the data model is depositing output data as a stream of events. Each event contains spectral information about a set of levels for a given simulation time step. Two kinds of components are pictured: interface objects and *concentrators*. The interface objects (X and Y in Figure 4.4) are equivalent to the object A (Figure 4.3), each one serving one Data_Object's user. Object fragments Z and T act like *concentrators*, meaning that they temporarily store and manage the data being produced by the model. The number and locality of concentrators can be dynamically configured, and they can serve multiple interface objects.

4.4.1 Data_Object's Design

In this section we present the basic objects and configuration entities that comprise the Data_Object's design.

Data_Object is a complex object composed by the following components: a *data_retriever*, a *transformer*, an *application_interface*, and a (potentially empty) set of *data_receiver* objects. Each one of these components is a configurable object

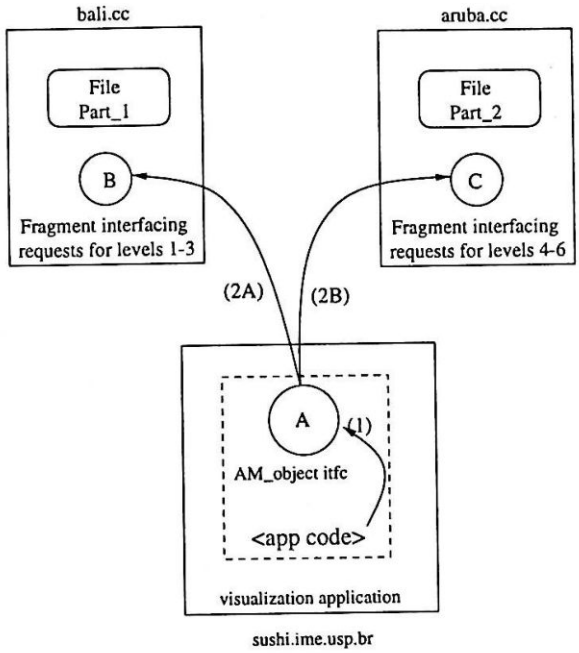

Figure 4.3: Data_Object obtaining input from files

implemented via objects, configuration objects, and configuration channels. The components may be distributed across computing, storage, and visualization[2] nodes. This distribution can be altered at runtime. Some of these components are fragmented themselves, increasing the opportunities for configuring object distribution. In addition to these object parts, *filters* may be dynamically created and attached to any of the components.

Figure 4.5 depicts a logical view of the Data_Object compound. The dashed circles represent configurable objects, therefore including a hierarchy of objects and configuration objects. The double-lined circles depict configuration entities that

[2]Visualization tools are just one example of data access applications. The Data_Object has been designed to work with the visualization tool implemented by our group, but it is not restricted to it.

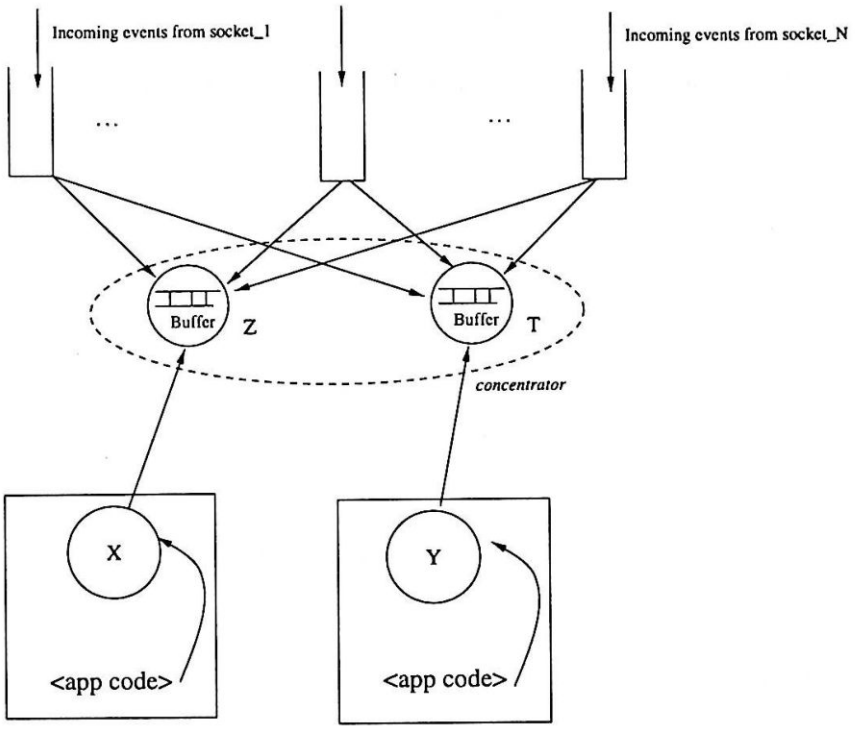

Figure 4.4: Data_Object layout for data obtained from running model

may be dynamically attached to the existing configurable objects.

The *data_retriever* abstracts the data sources. Usually it is configured at initialization time to access either files or sockets. The data is distributed in terms of atmospheric levels and/or time steps. The distribution can change at runtime (*e.g.*, more time steps are made available in new files).

The *transformer* object is responsible for performing the spectral to grid computation. It receives data from the *data_retriever* fragments and returns the associated grid points to the *application_interface* object.

The *application_interface* receives requests from the tool(s). It coordinates the

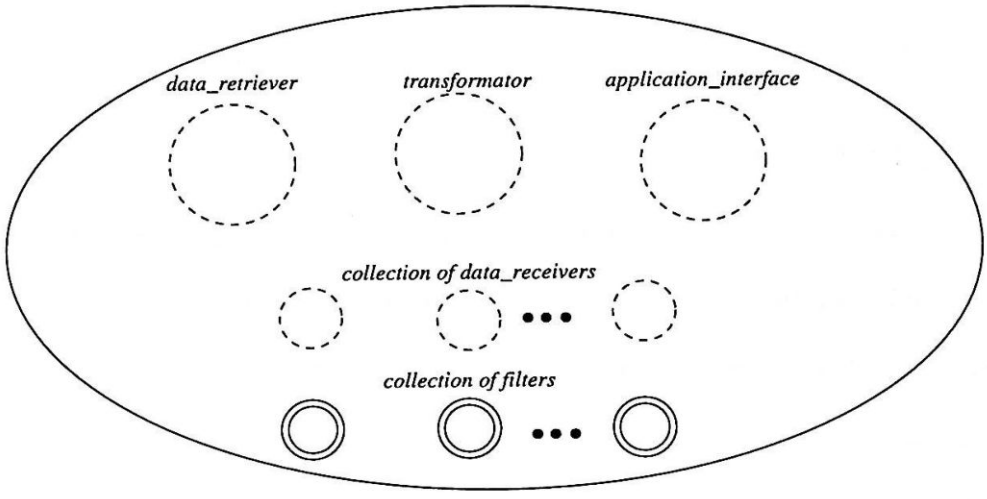

Figure 4.5: Data_Object components

components' behavior in order to retrieve, transform, and filter data.

Filters are configuration entities that may be attached to any of the Data_Object's components via configuration channels. They encapsulate a filtering function that is automatically applied to data arriving through the channel. *Filters* that support with sets of grid point values and generic sequences of numbers are available.

Finally, the *data_receiver* is another type of object through which a tool is able to obtain application data. It has been designed to provide simple support for propagating the accessed grid point data to a collection of cooperating agents. When a *data_receiver* is created, it is connected to the *application_interface* object, which will forward through the connection all data received. Connections can be temporarily closed by either the *data_receiver* or the *application_interface* object. Using *data_receiver* objects, the teacher-student pattern of collaboration can be achieved.

Filters can be attached separately to each *data_receiver*, thereby customizing the "student view".

Figures 4.6 and 4.7 depict possible ways in which the Data_Object parts may be distributed. In Figure 4.6, the data is read from files, the transformation from spectral to grid points is computed within the storing nodes, and no data filtering is performed. The *application_interface* object contains a *distributor* configuration object that forwards data requests to the appropriate *data_retriever* fragment. Figure 4.7 portrays a more complex *data_retriever*: the input data arrives through sockets and it is manipulated by the concentrators, which have attached filters that act upon the data before it is sent to the visualization node. By specifying attribute values, Data_Object's users can adapt it to different scenarios. For example, it is possible to transform the object setup described in Figure 4.6 into the arrangement in Figure 4.7.

The *Interface Description Language* (IDL) has been used to specify interfaces for the Data_Object and its components. Also, IDL types and manipulation interfaces have been defined for the spectral and grid point domains.

4.4.2 Data_Object's Creation and Adaptation

Two approaches for creating a Data_Object are available: (1) via a graphic user interface or (2) directly through the "COBS_object_creation" call (Appendix A).

In the first approach, the user provides information about the data sources and

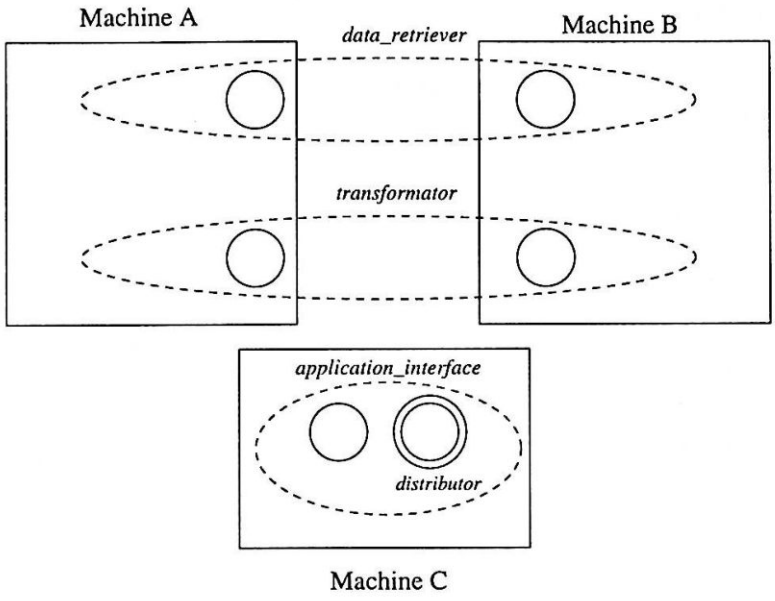

Figure 4.6: Example of Data_Object configuration

Data_Object's component distribution using buttons and menus. The interface prototype has been built using Tcl/Tk. It interprets the values provided by the user and defines the corresponding Data_Object's attributes. It verifies the given characteristics (*e.g.,* checks for file existence, socket connections, host names, object redefinition, *etc.*), invokes the COBS^{OM} object creation call, and registers the new Data_Object with the Object_Server. Tools can retrieve a Data_Object *proxy* from the Object_Server, and use it either for requesting data (as in the "get_grid" call) or connecting to a *data_receiver* that passively will receive the data retrieved from the Data_Object.

Alternatively, the COBS^{OM} object creation call can be directly invoked, with the data source description passed via attributes and arguments. Return values were

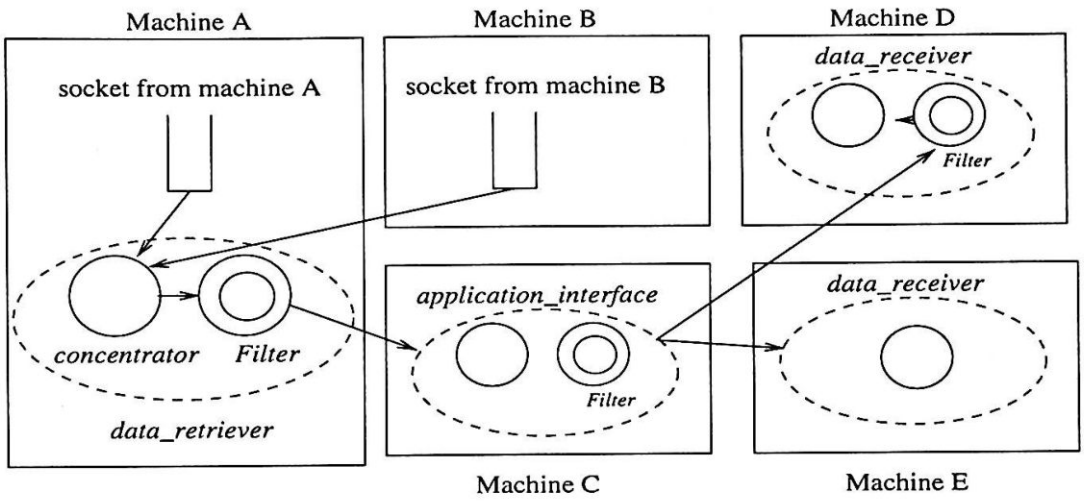

Figure 4.7: Another example of Data_Object configuration

defined for describing the most common situations in which creation may fail.

In our current implementation, COBS^{OTL}'s support for remote invocation and object identification is used. COBS^{OTL} provides primitives for creating references that can be invoked remotely, but these references can only be created in the local node, *i.e.*, the node in which the creation request is issued. In order to create the Data_Object's components on different nodes, we have designed Object_Factory, an object responsible for creating the objects and configuration entities needed by the Data_Object (*i.e.*, *data_retriever*, *transformer*, *application_interface*, *data_receiver*, *filters*, *distributor*, *concentrator*, *etc.*). It is necessary to have one Object_Factory running on each node involved in the Data_Object's fragmentation. These objects are registered in the Object_Server, which offers methods to retrieve object information based on host names.

Data_Object can be configured at runtime by attribute values passed through method invocations. For example, at creation time attributes can be used to specify the type of *data_retriever* to be created, input data characteristics, and distribution of *transformers* across nodes. Whenever Data_Object's methods are invoked, attributes can be passed to the configurable object resulting in adaptations such as *transformer* migration, attachment of new filters to object components, or disconnection of a *data_receiver*.

Some of the possible adaptations involve information better expressed through complex data types. For example, while adapting a *concentrator* (configuration entity related to *data_retriever* objects) from a central node to a distributed implementation — in which fragmentation format is derived from level information —, the distribution description can be naturally represented by an arbitrarily sized sequence of structures. The current COBS^{OTL} attribute implementation does not provide support for complex types. In order to spare the Data_Object's users from the cumbersome task of flattening the complex data structure into a collection of basic attributes, the Data_Object and its components' interfaces have been extended with a set of specific configuration methods. These methods receive configuration descriptions as arguments and change the appropriate attribute values and configuration channels.

CHAPTER V

RELATED WORK

Chapters 2 and 3 presented a concise overview of related work in configurable operating systems, concurrent programming, and distributed systems. In this chapter we describe related work in more detail. In many cases comparison between approaches in the literature and concepts/techniques in our work was carried out when such elements were introduced, and therefore will not be repeated here.

Section 5.1 describes the approaches for developing flexible systems from the object oriented programming community. Section 5.2 focuses on research on flexible and extensible operating systems. Section 5.3 discusses related research on configuring distributed systems and Section 5.4 summarizes this chapter.

5.1 Configuration in Object Oriented Programming

In [65], Pattie Maes introduces the notion of computational reflection to object oriented systems. Reflection had appeared before in the literature as a means to

describe a general theory and mechanism for allowing computational systems to reason about their won operations and structures[103]. Maes' paper presented examples showing that some programming problems that were previously handled on an ad hoc basis, can in a reflective architecture be solved more elegantly. Computational reflection is defined as the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation. A new concept (or programming-construct) is introduced: the notion of a meta-object. Meta-objects are just like the other objects of the language, except that they represent information about the computation performed by other objects and that they are also taken into account by the interpreter of the language (or its runtime support) when running a system.

Computational reflection is used in the Apertos operating system[112] to construct an operating system in an open and mobile computing environment. Objects appear as containers for information, while meta-objects define object's behavior semantics. A group of meta-objects compose a virtual machine. Object migration is the basic mechanism provided for accommodating heterogeneity in a mobile environment. The paper describes $m^{\text{BaseObject}}$, a "reflector" that provides facilities for concurrent object oriented programming; it offers the methods Call, Send, Reply, New, Delete, Grow, Shrink, Yield, Find, Install, and Migrate. In other words, it encapsulates the very basic runtime support for an object oriented system. $m^{\text{BaseObject}}$ provides both synchronous and asynchronous communication with remote procedure call semantics. Two examples of adaptation are presented in the paper: (1) persistence is added to an object by migrating it to a group of meta-objects that supports

persistent objects and (2) communication protocol is changed when moving through networks. Likewise COBS^{OM}, the Apertos' framework provides object/meta-object separation and meta-hierarchy.

Danforth, Forman, and Madduri presented their work on meta-objects in [37] and [24]. The papers describes how meta-objects are present in SOM (IBM System Object Model). The SOM runtime system supports the model and allows programs written in arbitrary languages to use the model via the SOM API. SOM is predicated on binary compatibility with respect to changes in class implementations. In both papers the notion of meta-class represents the class of a class. A class is different from ordinary objects because a class has an instance method table as part of its instance data. The instance method table defines the methods to which instances of the class respond.

In [37], the problem of composing different Before/After meta-classes in SOM is introduced and solved. A Before/After meta-class has a "before method" and an "after method" that are executed before and after the methods of the instances (classes) of its instances (objects). This abstraction can be used to implement method tracing, invariant checking, path expressing checking, object locking, etc. Problems appear when multiple inheritance is used, since the Before/After methods of the involved parent classes have to be combined. The solution presented in the paper does a pre-order traversal of the meta-class hierarchy, looking for the first meta-class on each path that defines a Before-Method; each such Before-Method is then invoked. Only one Before-Method is called in each path, therefore the designer

of a Before/After meta-class has to know the parents of this meta-class and explicitly invoke any inherited functionality it needs. Every Before/After meta-class is derived from the SOMclass. This basis class provides the somDispatch method, and a class can arrange its instance method table so that all method calls are routed through somDispatch. Notice that the effect is similar to the invocation interception by policy objects described in Chapter 2.

The second paper[24], by Danforth and Forman, reports on the evolution of meta-class programming in SOM. It presents SOM as (1) a way to minimize recompilation of binaries when libraries are replaced and (2) an example of reflection usage (SOM is used to implement SOM). The paper describes the evolution of SOM (version 1.0 to 2.0 and 2.1) and focuses on solving the problem of using two meta-classes that both redefine some basic method for creation of instances or initialization of classes. The problem is solved via a "cooperative framework", which is a set of functions that the programmer can use to register his/her own functions as the new methods overrides, but in such a way that the original method for the class is created too. This is roughly equivalent to our use of policy objects' initialization code, which runs before the initialization code for the basic object.

Mohindra, Copeland, and Devarakonda present a dynamic approach to adding services to objects and compare it to the static approach. They approach the problem using dynamic subclassing and a meta-class that allows the insertion of a *before* and an *after* method around each method of the original class.

5.2 Configuration in Operating Systems

Many researches have worked on improving the flexibility of services provided by operating systems. Dynamic linking does not solve the problem because it tightly couples clients to a particular service implementation and does not provide for transparent routing of requests to alternate or supplementary services.

The Spring operating system[45, 53, 75] introduced *subcontracts* as a flexible base for distributed computing. The motivation was to permit application programmers to control fundamental object mechanisms in the context of an object-oriented distributed system. The framework provided by subcontracts makes it easy for different remote procedure call mechanisms to work together, and for implementors to add new mechanisms in a consistent, compatible way. Subcontracts are based on some early experience with CORBA object adaptors, which are elements that provide a (limited) set of choices about the server-side object mechanism. Subcontracts are part of the Spring object model. A Spring object is perceived by a client as consisting of three things: a method table, a subcontract operations vector, and the object representation (some client-local private state). Code generated automatically from an IDL interface description transforms method invocations into calls on either the object's regular method table or on its subcontract operations vector. How the subcontract's methods achieve their effect is hidden from the client. The main client-side subcontract operations are marshal, invoke, unmarshal, marshal_copy, and invoke_preamble. They represent the opportunities that the subcontract has to

act upon the invocation. For example, "invoke_preamble" allows an object's subcontract to become involved in the invocation early in the process. The subcontract "invoke" operation is only executed after all the argument marshaling has already occurred.

The server-side interfaces can vary considerably between subcontracts. There are three elements that are typically present: support for creating a Spring object from a language-level object, support for processing incoming calls, and support for revoking an object. A subcontract is involved in all the key events of the object's life: its birth, its reproduction, its death, its transfer between address spaces and whenever any invocations occurred on the object. Subcontracts separates out the business of implementing objects from implementing object mechanisms. It is presented as a effective way for plugging in different policies for different objects. Similar to the COBS^{OM}configuration model, Spring introduces the notion of compatible subcontracts: a subcontract A is said to be compatible with subcontract B if the unmarshaling code for subcontract B can correctly cope with receiving an object of subcontract A. In [53], the development of an extensible file system in Spring is described. A key idea in this work is that the memory object is decoupled from the pager. The proposed scheme works as a building block for implementing caching and coherency: subtyping interface capability makes things cleaner in terms of putting new file systems on top of others. A file system object can be at same domain or different domains (on same or remote machine). This object can be created dynamically. They have *context objects* to implement a flexible name space. In [75], an unified caching architecture is presented. As the paper abstract explains,

the unified caching scheme can be used to cache a variety of different kinds of remote objects. For any given kind of object, the scheme lets different client processes within a single machine to keep a single cache for accessing remote objects. This caching is performed by a separate cacher process on the machine local to client processes; the caching is transparent to the clients, and the cached information is kept coherent. This architecture has been used to implement caching for files and for naming contexts.

In summary, subcontracts offer methods that can be invoked by stubs or by the code for creating, copying, or deleting objects. These are features that can modify the behavior of basic object mechanisms. Subcontracts differ from our model for configurable objects in many ways:

- subcontracts are used by the object model, not by the basic object implementor. For the programmer, all that is available is the choice of which subcontract to use with a type. This specifies a "flavor" for the object, since it defines how the object support mechanisms will run. Once an object is created, application programmers can not change the subcontract associated with it;
- only one subcontract can be associated with a given object; and
- the interaction pattern between type and subcontract is fixed. For example, the application can not make the subcontract to skip "invoke". Only the subcontract itself can control its methods.

The Synthesis Kernel[82, 70] is a distributed operating systems that adapts the

code in order to improve performance. In this work, it was demonstrated that performance gains at execution time can outweigh code generation costs in a hand-tuned kernel. The system combines efficient kernel calls with a high-level, orthogonal interface. The system provides a simple computational model that supports parallel and distributed processing. The model's interface consists of six operations on four kinds of objects. Pu and Massalin's research investigated the use of a code synthesizer in the kernel to generate specialized (thus short and fast) kernel routines for specific situations. For example, typical Synthesis routines contain from 20 to 30 machine instructions, while the 4.3 BSD read call contains about 500 lines of C code. They employed three methods to synthesize code: (1) factoring invariants to bypass redundant computations, (2) collapsing layers to eliminate the unnecessary procedure calls and context switches; and (3) executable data structures, which are data structures extended with executable code so they can be self-traversing, thereby decreasing the transversal overhead. More recently, Pu's work has been exploring the use of incremental partial evaluation for achieving high performance modular portable operating systems[19]. The idea is to progressively specialize the implementation of each operation as its ultimate execution context is exposed. In [81], a technique is described for generating specialized kernel code optimistically for system states that are likely to occur, but not certain. Program specialization is usually performed in terms of available invariants, *i.e.,* bindings that are known to be constant. Examples of invariants in operating systems available before runtime include processor cache size, whether there is a float-pointing unit, etc. In operating systems, there are many things that are likely to be constant for long

periods of time, but may occasionally vary. An example from the paper is that files are not likely to be shared concurrently, and that all reads to a particular file are likely to be sequential. They call these assumptions "quasi-invariants" and generate code specialized for these quasi-variant scenarios. Correctness can be preserved by guarding every place where quasi-invariants may become false. If one or more assumptions become false, the routine version is replaced by another (in a different stage of specialization). The paper [21] discusses techniques for fast concurrent dynamic linking, thereby alloying efficient replacement of versions. One important aspect of using program specialization in the operating system context is that the full cost of code generation at runtime may be too high to be afforded. This cost can be avoided by generating "code templates" statically and optimistically at compile time. At kernel call invocation time, the templates are filled in and bound appropriately. These ideas were applied in a real-world operating system by re-implementing the HP-UX file system. The experiments showed that the specialized read system call reduces the cost of a single byte read by a factor of 3, and an 8 KB read by 26%, while preserving the semantics of the HP-UX read call. By relaxing the semantics of HP-UX read, Pu et al were able to cut the cost of a single byte read system call by more than an order of magnitude.

In the work by Pardyak and Bershad[79], an event-based invocation mechanism is used to provide flexible, transparent, safe, and efficient dynamic binding of extensions. By installing a handler on an event, an extension's code can execute in response to activities at the granularity of the procedure call. Handlers are dynamically added to or removed from an event, and any number of them can be installed

at any particular time. Their work rely on a set of static and dynamic access control mechanisms to ensure that events are not misused by extensions. Events can run synchronously or asynchronously, but SPIN[8] only allows the use of asynchrony where it would not violate the semantics of a procedure call. This approach, differently from ours, is tightly coupled with the compiler (Modula-3) runtime system.

Jones' work on interposition agents[52] offers configurability of system calls by providing a toolkit for transparently interposing user code at the system interface. Jones' toolkit allows the code to be interposed between clients and instances of the system interface to be written in terms of the high-level objects provided by this interface, rather than in terms of the intercepted system calls themselves. The toolkit has been used to construct system call tracing tools, file reference tracing tools, and customizable file system views. The toolkit presents the basic operating system abstractions (pathnames, descriptors, processes, process groups, files, directories, symbolic links, pipes, sockets, signals, devices, users, groups, permissions, time) in an object-oriented programming language. The classes provided by the interposition toolkit offer a hierarchy where the lowest layers perform such functions as agent invocations, system call interception, and incoming signal handling. Above this layer are available objects to resolve pathnames, file descriptor name space, and reference counted open objects. The third layer focuses on secondary objects provided by the system call interface (normally accessed via primary objects), such as files, directories, symbolic links, devices, pipes, and sockets. The goal in this work was to be able add user code without modifying the system or the kernel. This contrasts with our approach, in which our pursuit for flexibility may imply in redesigning basic objects

and separating basic functionality from configurable issues related to performance.

As part of his PhD research, Peter Druschel investigated efficient support for incremental customization of operating system services[25]. The main idea is to achieve efficient, incremental customization of operating system services using a two-fold strategy: (1) an object-oriented architecture that relies on composition to facilitate code reuse and customization and (2) an operating system structure that places a minimal set of trusted functions into the kernel, with all remaining services collocated with application code in user-level protection domains. His ideas have been explored in the context of Lipto, a composable object-oriented operating systems[26]. An interesting aspect of their use of object-oriented architecture is that composition is used instead of subclass specialization. Objects are composed, or pieced together, to form services. A service implementation can be modified by composing it with a different set of objects. The replacement of constituent objects that offer standard services is made using a secondary, meta-level interface. This meta-level interface includes interface definitions for all the constituent objects, and a mechanism for composing a service by specifying a graph that binds the constituent objects. Conventional meta-level interfaces usually present a problem: the use of application-provided object implementations introduces communication overheads which may offset the performance advantages of customization. The reason is that invocations of a service usually must cross a domain boundary. The solution is to decompose the operating system implementation in two parts. The first is a minimal set of functions that must be trusted and centralized. This typically includes code that mediates access to (and use of) physical and other limited resources. The

approach can only be successful if it is possible to decompose operating services in a way that the set of trusted, centralized functions is small.

Lipto[26] proposes an architecture that provides location transparent binding and access of modules optimized for the local case, thereby decoupling the orthogonal concepts of modularity and protection. Given such support, the decomposition of a system into modules can be guided by sound software engineering principles, with the modules distributed across protection domains and machines at configuration time based on criteria such as trust, security, fault-tolerance, and performance. The partitioning of modules into domains can be adjusted according to their state in the software life-cycle, and/or the requirements in a particular installation of the system. For instance, a subsystem consisting of a set of modules can be configured with each of its modules in a separate domain for ease of fault detection and debugging during its testing phase, and later, the post-release phase, combined into a single domain (*e.g.,* the kernel domain) for performance. Druschel, Peterson, and Hutchinson concluded from their experience with the x-kernel[48] that a system built in such a manner can be extended vertically through layered services without imposing cross-domain invocation costs at each layer. Runtime configuration is used in their work to make non-local invocation performance efficient: their invocation mechanism relies on a configurable RPC service, which allows the dynamic selection of the most appropriate RPC mechanism[80].

Engler, Kaashoek, and O'Toole have proposed the Exokernel architecture[31],

where a small kernel securely multiplexes available hardware resources among applications. Library operating systems use this interface to implement system objects and policies. The separation of resource protection from management allows application-specific customization of traditional operating system abstractions by extending, specializing, or even replacing libraries. Instead of providing each application with its own virtual machine, the exokernel exports hardware resources rather than emulating them, thereby avoiding the severe performance penalties usually associated with virtual machines. They demonstrated the flexibility of the exokernel architecture by showing how fundamental operating system abstractions can be redefined by simply changing application-level libraries. They built extensions to RPC, page-table structures, and schedulers. They show that an exokernel can be implemented efficiently by evaluating the performance of two prototypes : Aegis, an exokernel, and ExOS, a library operating system. ExOS manages fundamental operating system abstractions (*e.g.,* virtual memory and process) at application level, completely within the address space of the application.

Tanenbaum, Doorn, and Homburg proposed an extensible kernel, called Paramecium[108]. This kernel uses an object-based software architecture which together with instance naming, late binding and explicit overrides enable easy reconfiguration. Determining which components reside in the kernel protection domain is up to the user. In the Paramecium architecture an object is conceptually a collection of methods and instance data. Each object exports one or more named interfaces, thereby providing support for evolving and generic interfaces. Objects are relatively coarse grained. They can contain operating system components such

as a scheduler, IP layer, device driver, or application components such as memory allocators or matrices. Method delegation is supported. The system architecture consists of a nucleus and a repository of system components. The nucleus is a protected and trusted component that implement the services that can not be trusted to the application: processor event management, memory management, directory service, and certification service.

FLEX[12] is a tool for building efficient and flexible systems that is being built by researchers from University of Utah. The tool has been designed to provide support for powerful operating systems interface efficiently by constructing specialized module implementations at runtime. FLEX is a coarse-grain system building service that allows systems to be dynamically constructed using the implementation most appropriate to a given situation. The tool can dynamically extend the kernel in a controlled fashion, which gives user programs access to privileged data and devices not envisioned by the original operating system implementor. Operating system modifications are done on the fly, by biding routines to the kernel on a per-client basis. FLEX works by reading modules, manipulating them as specified by manipulation files, and writing the resulting executable either to a user's address space or to a file. In addition, FLEX lets kernel-level services and associated trap vectors to be added by the user.

Arindam Banerji proposed an infrastructure for extending operating systems[6]: protected shared libraries, a new form of modularity that can be the basis for building flexible and library-based operating system services. It extends the familiar

notion of UNIX shared libraries by adding protected state and cross protection boundary data sharing. The protected shared libraries infrastructure has been integrated into a existing operating system (IBM AIX 3.2.5). In [4, 5], Banerji and Cohn introduced two frameworks architectures for flexibility and composability : substrates and aggregates. Substrates have been designed to capture reusable patterns and solutions for building flexible software, whereas aggregates incorporate the mechanisms for composing software solutions. High-level specification of an operating system component is generated as a framework that incorporates the substrate architecture. Similarly, description of the interaction between such components is compiled into an aggregate framework that captures at run-time the behavioral dependencies between implementation constituents. Substrates are envisioned as mix-ins for system services that may be aggregated to form complex subsystems. A substrate is a three-dimensional object that encapsulates a particular operating system functionality. Each dimension of the framework-based realization of a substrate presents to the client a different interface or view of the encapsulated functionality. The primary interface provides the main functionality of a substrate. The secondary interface opens up the implementation of the substrate through domain-specific hints and directives. The tertiary interface presents a modifiable self-representation of the meta-mechanisms of the substrate object system. The support for the programming style of building substrates and aggregating them has been incorporated into the AIX 3.2 kernel. Several optimizations to boost the performance of substrates have also been done. Dynamically loadable system call classes allow AIX kernel services

to be presented as C++ interfaces to user level and kernel clients alike. These interfaces promote cross-domain use and specialization of system services. The substrate programming environment was closely modeled after the CORBA specification[77].

Kea[109] is an operating system kernel designed for maximum flexibility and performance in the areas of dynamic reconfiguration and extensibility, both from the kernel and application viewpoints. Kea's form of RPC is general and allows the destination of an RPC to be changed independently of the caller. Inter-domain entry points can be remapped, thereby allowing systems to be reconfigured.

5.3 Configuration in Distributed Systems

Distributed systems can be perceived as a collection of modules and interconnections. Some of the interconnections are created at run-time. Many researchers have been addressing the problem of changing interconnection configuration, *i.e.*, carrying out modifications on the structure of communication among the system components that may occur while these modules are executing and interacting. The Conic[58] system provides a graphical user interface for configuring the structure of interconnections. Warren and Sommervile presents a synchronization mechanism that allows dynamic configuration while preserving application integrity[110]. Argus[62] supports reconfiguration with two phase locking over atomic objects and version management recovery techniques.

Magee, Dulay, and Kramer have developed a constructive development environment for parallel and distributed programming[67]: Regis. Regis evolved from Conic and REX. It perceives a parallel or distributed application as a collection of components with complex interconnection patterns. Communication is separated from computation, both parts being expressed through pieces of code written in C++. The program configuration — program description in terms of how its components related to each other — is specified using the language Darwin. Regis supports the development of programs in which the process structure changes as execution proceeds. This is achieved through two mechanisms: (1) lazy instantiation, which useful for recursive structures, and (2) dynamic instantiation, *i.e.,* components are created dynamically as they are needed.

Swaminathan and Goldman propose a solution to modifying the communication pattern in distributed applications. The solution does not violate the usual (and desired) separation between communication structure and computation. Their abstractions provide for modifications on the communication patterns without requiring modules to know how the system is configured. They introduce a special data type (a "handle"[106]) that represents the module external ports. Connections can be established between handles.

The system Equus[55], developed by Kindberg, Sahiner, and Paker, supports adaptive parallelism. The computation execute on a processor pool, expanding and contracting as the number of processor nodes allocated to them varies over its run-time. Computations are based upon hierarchical master-worker structures. The

number of worker processes changes with the number of allocated nodes, and so does the number of processes that act as servers to them. The paper [55] presents an example from the image-processing domain, describing the synchronization between workers and processes, changes in communication linkages, and how in some cases state is transfered between them. Reconfigurations are transparent to the worker processes, but not all types of adaptations can be made transparent to servers.

In addition to the work in the Paramecium system (Section 5.2), Tanenbaum, Homburg, Doorn, van Steen and de Jonge propose an object model for flexible distributed systems[47]. Their goal is to provide high-level primitives for supporting distributed and parallel application. They provide flexibility in terms of allowing the configuration of both applications and kernels so that they only include the functionality that is actually used. Therefore, their approach differs from ours in the extent in which flexibility is pursued. In their work, flexibility is provided by allowing extensions of operating systems kernel and applications with new objects at run time, and by providing a way to bind to objects dynamically. (The process of installing and initializing such objects in an address space is called binding). A distributed object is pictured as an object that can have interface instances in multiple address spaces or is fragmented over multiple address spaces. Method invocation on a distributed object is only possible in a given address space if that address space contains a local object that is part of the distributed object. This contrasts with the distributed objects in COBS^{OM}, where remote invocation requires only object ids which can be retrieved via the Object_Server(Section 4.3).

Configuration issues have been addressed by many researches from the fault tolerance community[7]. The system Sampa[30] provides support for the management of fault-tolerant distributed programs according to user provided and application-specific availability specifications, with some reconfiguration and recovery actions being automated by the system. The system architecture is based on "agents" and "supervisors". Agents residing on every host node and execute process management tasks and monitoring. A supervisor makes global management decisions about the distributed services according to availability specifications and information provided by the agents.

Little and McCue's work[63, 64] explored configuration in the context of managing a replica system. In their system, built on top of Arjuna[100] — a programming system for reliable distributed computing —, the consistency protocol being used can be configured. The system allows the applications to control number and location of replicas and use application specific knowledge about object inter-dependencies.

5.4 Summary

The development of object oriented technology incited new ways of structuring implementations. It became common sense that in some arenas (*e.g.,* operating systems, real time systems) many implementation decisions did not represent "just details"; on the contrary, they had a crucial impact on performance ([54], [16],

[99]). Object orientation has been proposed as a clean way of allowing incremental design, robustness and incorporating diversity of implementation alternatives. In particular, object oriented languages that directly support the use of *reflective programming* ([65]) and *meta-objects* ([37], [24]) such as Smalltalk and CLOS are advocated by researchers in the object oriented community as ideal environments for developing flexible systems. David Shang proposed and implemented a object oriented framework where flexibility is attained by dynamic instantiation of parametric classes, focusing mainly in installation time configuration[97]; it differs from traditional work on parameterized classes or functions by offering dynamic instantiation and dynamic binding.

The *Open implementation* approach has been proposed in the context of meta-objects and reflective programming[32], and evolved into the use of those ideas in environments where performance requirements prohibited the maintenance of a complete framework for interpreting and redefining object properties. In ACE[85]/ASX[86] the support for flexibility is explored through dynamic configuration of which services map onto which hosts in a distributed system and how available parallelism on multiprocessor platforms can be effectively used. Tigger [114] illustrates the use of *open implementation* to achieve customization of behavior of application-level objects in real-time environments.

Spring[45] introduces the concept of *subcontract* as a vehicle to achieve flexibility in implementing remote procedure calls, and the object oriented structuring of their system is showed to be suitable to achieve flexibility and uniformity in specialization

of services[53].

Banerji and Cohn propose an infrastructure for building highly modular and flexible operating system components that can be composed to provide a variety of application specific system services([4],[5]). The basic element in the framework is a *substrate*, which can be considered a three dimensional object where each dimension presents a different view of the encapsulated functionality: the primary interface offers principal functionality, the secondary accepts domain-specific hints and directives, and the tertiary presents modifiable representations of the substrate. The realization of a substrate is achieved by a partition in client-instance communication, instance-interface, and interface-implementation. These partitions share the "function evaluation framework", *i.e.,* a meta-level way to specify member functions, canonical functional call formats, and dispatches tables.

Interesting results have been achieved by Synthesis([69], [81]) by specializing critical fragments of code. Good results by using the interposition technique were reported in [52].

Interesting insights about how to compose configurable distributed applications are present in Polylith[1], Regis[67], Equus[55].

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

The *Configuration Toolkit* (CTK) described and evaluated in this thesis 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 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 enables 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 permit the online configuration of object performance or reliability to match current application needs. As a result, CTK is shown useful for implementing the complex adaptive object behaviors required for high performance programs running on modern 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 thesis demonstrated that performance improvements derived from the runtime 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 application programs, we expect to achieve cumulative performance gains approaching or exceeding 100%. Similarly, our current work with the runtime configuration of objects on distributed platforms indicates that substantial performance gains can be attained by configuring applications simultaneously at several levels of abstraction in the distributed program and in the underlying operating systems and networks[40]: (1) in the application itself, (2) in the object transport substrate, and (3) at the protocol level. Our previous work has demonstrated the performance gains from configuration at each of these levels[49, 40], but we have not yet been able to show gains by simultaneous configuration at multiple levels of abstraction.

The extensions of object functionality described in this thesis 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.

We introduced the notion of 'configurable objects'. They describe the association of policies with objects as first class objects themselves, and by implementing object functionality using CORBA-compliant runtime libraries and compilation support. Namely, we have developed a prototype of the framework within which object implementations can range from 'memory' objects permitting programs to share unstructured, raw data stored in memory pages, to typed and structured objects explicitly defined by application programs. Moreover, the implementations of such objects can take advantage of underlying platforms offering multiple standard and custom communication protocols, and they can utilize the performance techniques required for their needs, including replication, caching, fragmentation, and dynamic adaptation of selected aspects of their implementations.

In summary, the framework presented in this thesis provides a programming model and environment where flexible software can be developed by designing configurable objects. COBS^{OM}'s 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, by explicitly separating the type-dependent object functionality from its properties subject to configuration, including its performance, reliability, and timing properties. *Objects* and *configuration objects* can be combined in complex

ways, and the composition can be changed dynamically without the imposition of unreasonable overheads. Our experience in building configurable objects, in particular the Data_Object presented in this thesis, indicates that COBS^{OM} is suitable for building high performance distributed and parallel objects that can achieve flexible and complex behavior in runtime. Moreover, the programming model offered by COBS^{OM} encourages incremental design and reuse of components.

As part of future research, we intend to evaluate the object technologies described in this thesis in the context of highly dynamic distributed applications. 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 thesis to develop new technologies for online program configuration.

APPENDIX A

COBS_OBJECT_MODEL INTERFACE

This appendix presents a short description of the COBS^{OM} interface. Section A.1 lists the routines available to manipulate objects and configuration objects. Section A.2 describes how the COBS^{OM} framework can be used to implement new objects: the necessary steps to associate implementation code with an IDL interface are enumerated and an example is presented. Finally, Section A.3 lists some important internal routines.

A.1 COBS^{OM}'s Interface

The programmer implementing COBS^{OM} objects should use the provided interface for object creation, method invocation, configuration entities manipulation, and configuration channel operation.

For every interface ltfc being used by an application, COBS^{OM} generates the following routines:

1. RESULT COBS_ltfc_object_create (ltfc *object*, attr_list *creation_attributes*)

Common values for "creation_attributes" are attributes for naming the object or specifying that it should be created as a "remote object", *i.e.*, an object that can be invoked from other nodes.

2. for every method Mthd (which returns a ReturnType value) specified in the Itfc definition, the following is defined:

   (a) ReturnType COBS_invoke_Itfc_Mthd (Itfc *object*,
   CORBA_Environment *ev,
   < argument_list >)

   (b) a macro "COBS_method_id_Itfc_Mthd", which represents an integer that uniquely identifies Mthd among Itfc's methods.

For every configuration entity Cfg, the following routine is available:

RESULT COBS_CONF_create_Cfg (COBS_Conf_Object *cob)

For every configuration channel Chn described in Cfg, the macro COBS_channel_id_Cfg_Chn is available for channel identification. Similarly, for every required method ReqMeth present in the Cfg specification, the macro "COBS_CONF_req_method_id_Cfg_ReqMeth" is available. A required method whose return type is ReturnTypecan be activated by the following routine:

ReturnType COBS_CONF_invoke_req_method_Cfg_ReqMeth (
COBS_CONF_Object_t *cobj*,
COBS_Object_t ob, CORBA_Environment *ev,

< argument_list >)

The routine for binding configuration entities to objects has the following interface:

```
RESULT COBS_bind_configuration (COBS_Object *o,
                                COBS_Itfc_Description *class,
                                COBS_Conf_Object *conf,
                                COBS_Conf_Description *conf_desc,
                                char binding_kind );
```

The routine for opening configuration channels is

```
RESULT COBS_open_conf_channel (COBS_Object_t obj,
                               int obj_method_id,
                               COBS_Conf_Object_t conf_entity,
                               COBS_Conf_Description *conf_desc,
                               int conf_channel_id );
```

Two routines are available for querying the Interface Repository:

```
RESULT COBS_get_Itfc_Description (COBS_Itfc_Description **desc,
                                  char *itfc_name, char *directory );
RESULT COBS_get_Conf_Description (COBS_Conf_Description **desc,
                                  char *conf_name, char *directory);
```

For programming distributed applications, the following routines are useful:

void COBS_init_app(char **app_name*);

RESULT COBS_get_object_server(CORBA_Object **obj_server*));

We provide some support for manipulating IDL complex types:

char *COBS_return_string(char **st*, int *st_maximum*);
void COBS_assign_string(char ***stto*, char **stfrom*, int *st_maximum*);
void COBS_init_sequence (COBS_sequence_struct **seq*,
long *maximum*, int *basetypesize*);
void COBS_assign_sequence (COBS_sequence_struct **left_side*,
COBS_sequence_struct **right_side*,
int *basetypesize*);

A.2 Implementing COBS^{OM} objects

COBS^{OM} requires the object implementor to follow some guidelines when coding the modules that implement IDL interfaces.

Let ltfc be an IDL interface specification that offers the methods $m_1, m_2, \ldots, m_n$.

The implementation module for ltfc has to include the following information[1]:

1. files to be included

   #include "config.h"

   #include "ltfc_COBS_.h"

2. data structure that represents the object state

   It has to be named "ltfc_state_struct". At some point after the structure has been defined, the file "ltfc_STATE_COBS_.h" has to be included.

   The macro "ltfc_STATE(object,struct_field)" is defined for accessing a field in the object instance data structure.

3. code for methods $m_1, \ldots, m_n$

   For each method Mthd (with return type ReturnType) specified in the interface, the corresponding C function should be defined as follows:

   ReturnType ltfc_Mthd (ltfc *object*, CORBA_Environment **ev*, <*arg_list*>)

   The *arg_list* represents the list of arguments being passed to the method. The implementor should follow the directives in the CORBA document[77] for C language stub mapping. Table A.1 summarizes the guidelines for passing arguments to a stub and receiving values as a result of the method execution.

[1]The string ltfc should be substitute for the actual interface name being used.

IDL type	Pass In	Pass Out/InOut	Result
short	value	addr of var to hold val	receive value
long	value	addr of var to hold val	receive value
unsigned short	value	addr of var to hold val	receive value
unsigned long	value	addr of var to hold val	receive value
float	value	addr of var to hold val	receive value
double	value	addr of var to hold val	receive value
boolean	value	addr of var to hold val	receive value
char	value	addr of var to hold val	receive value
octet	value	addr of var to hold val	receive value
enumeration	value	addr of var to hold val	receive value
struct	addr of struc	addr of var to hold struct	receive val of struct
union	addr of struc	addr of var to hold struct	receive val of struct
string	ad 1st char	addr of (char *)var	receive char*
sequence	ad seq. struc	addr of seq. struct	rcv val of seq. struc

Table A.1: Guidelines for Argument Passing and Return Values

Example of IDL specification and associated with implementation module

The following IDL specification is a simplification of the Object_Server interface:

```
typedef string          astring;
typedef long            aobjref;
interface obj_server {
  typedef sequence<aobjref> seq_frags;
  char init();
  char register_object(in astring app_name, in astring obj_name,
                       in astring hostname,
                       in aobjref obj_ref);
```

```
  char register_object_with_replacement(in astring app_name,
                                        in astring obj_name,
                                        in astring hostname,
                                        in aobjref obj_ref);
  char find_object (in astring app_name, in astring obj_name,
                    out aobjref obj_ref);
  char find_object_host_specific(in astring app_name,
                                 in astring obj_name,
                                 in astring hostname,
                                 out aobjref obj_ref);
  char register_fragmented_object(in astring app_name,
                                  in astring obj_name,
                                  in astring mainhostname,
                                  in aobjref obj_ref,
                                  in long total_frags,
                                  in long node_id,
                                  out seq_frags frags);
  void del_app(in astring app_name);
  long nb_apps();
  long nb_objects_in_app(in astring app_name);
  long nb_objects_total();
  void dump();
};
```

The following C code fragments are part of our implementation for the obj_server interface:

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include "config.h"
#include <otl_thread.h>
#include "IDLtoC.h"
#include  "obj_server_COBS_.h"
#define PRIME_NB      11
#define MAX_APPS      50
#define MAX_OBJECTS_PER_APP   100
/* Defining state */
typedef struct {
  mutex_t lock;
  volatile int counter;
  int nb_frags;
  aobjref *array_frags;
} FragInfo;
typedef struct {
  ...
} HashEntry;
```

```
typedef struct {
  ...
} HashTable;
typedef struct {
  mutex_t       object_server_lock;
  HashTable     app_hash;
  HashTable     **app_object_tables;
  long          nb_objs_total;
} obj_server_state_struct;
#include "obj_server_STATE_COBS_.h"
static int HashCreate(HashTable *hashtable, int nb_elems)
{ ... }
CORBA_char  obj_server_init (obj_server object, CORBA_Environment *ev)
{...}
CORBA_char  obj_server_register_object(obj_server object,
                                       CORBA_Environment *ev,
                                       astring app_name,
                                       astring obj_name,
                                       astring hostname,
                                       aobjref obj_ref)
{...}
CORBA_long obj_server_nb_objects_total(obj_server object,
                                       CORBA_Environment *ev)
```

```
{
  assert(object->state != NULL);a
  return(obj_server_STATE(object, nb_objs_total));
}
CORBA_void obj_server_dump(obj_server object, CORBA_Environment *ev)
{ ... }
```

A.3 Internal Routines

Many routines and data structures are generated from an interface description and used internally by COBS^{OM}. In this section, a list of some important internal information is presented. The understanding of these parts is required for changing and extending the library. This information is not used by programmers developing new COBS^{OM} objects.

For every interface Itfc, a data structure for the attributes defined in the IDL description is created. The information can be generated as either a "struct" or a COBS^{OTL}attribute list.

For every method Mthd in Itfc, a structure "_COBS_pb_Itfc_Mthd" is generated for the parameter block.

The following generated methods are employed during object creation and invocation:

```
RESULT _COBS_Itfc_Mthd_create_pb (_COBS_pb_Itfc_Mthd **)

void _COBS_through_pb_Itfc_Mthd(_COBS_pb_Itfc_Mthd *)

void _COBS_directly_through_pb_Itfc_Mthd(_COBS_pb_Itfc_Mthd *)

RESULT COBS_pb_alloc (memory_t *ptr, int size);

void _COBS_invoke_through_channels (Operation_Info op, any_t pb);

RESULT COBS_obj_create (COBS_Object_t *ob, int attrib_size,
                        int state_size,
                        int nb_of_operations, char *itfc_name);
RESULT COBS_create_conf_channel (Conf_channel_t *ch,
                                 COBS_Channel_Desc ch_desc);
int _COBS_CONF_find_obj_info (COBS_Conf_Object_t config_obj,
                              COBS_Object_t ob);
void _COBS_CONF_get_invoke_info (COBS_Conf_Object_t config_obj,
                                 int obj_id,
                                 int req_method_id,
                                 _COBS_create_pb_funcType *p,
                                 _COBS_invoke_function_funcType *f);
char _COBS_is_this_obj_local (CORBA_Object o);
```

APPENDIX B

FORMATS FOR DESCRIPTION

This appendix presents the input format supported by the tools code_gen, conf_gen, and app_gen. The grammar presented here (using the BNF format) could have been specified more clearly through a concise set of rules. Instead, the definitions mirror the current *lex/flex* and *yacc/bison* implementation files, which are still evolving.

B.1 Meta-description for object interfaces

The output produced by idl2c and consumed by code_gen has the following format:

<*description*> ::= $\{<$ *interface* $>\}^*$ $\{<$ *typedesc* $>\}^*$

<*interface*> ::= Interface <*name*> $\{<$ *attribute* $>\}^*$ $\{<$ *operation* $>\}^*$

<*attribute*> ::= Attribute <*name*> TypeName <*name*>
<*typeinfo*> $\{$Readonly$\}^?$

<*operation*> ::= Operation <*name*> TypeName
$\{<$ *argument_list* $>\}^?$ $\{<$ *context* $>\}^?$

<*context*> ::= Context $\{< name >\}^+$

<*argument_list*> ::= Argument <*name*> {IN | OUT | INOUT}

TypeName <*name*> <*typeinfo*>

<*typeinfo*> ::= TypeCode <*type*>

<*type*> ::= NT_enum | NT_interface

| NT_pre_defined | NT_struct | NT_array

| <*string*> | <*sequence*>

<*string*> ::= NT_string MAXLEN <*number*>

<*sequence*> ::= NT_sequence MAXLEN <*number*>

BaseTypeName <*name*>

BaseTypeCode <*basetcode*>

<*basetcode*> ::= NT_enum | NT_pre_defined

| NT_struct | NT_array

| NT_string | NT_sequence

<*typedesc*> ::= TYPEINFO <*desc*>

<*desc*> ::= <*name*> NT_struct NB_FIELDS

<*number*> $\{< field >\}^+$

| <*name*> <*string*>

| <*name*> <*sequence*>

| NT_array

| <*name*> TYPEDEF_OF <*name*>

<*field*> ::= FIELDTYPE <*name*> BaseTypeCode <*basetcode*>

FIELDNAME <*name*>

<*name*> ::= <*letter*> {< *letter* > | < *digit* > | '_' }*

<*letter*> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'

<*number*> ::= {< *digit* >}+

<*digit*> ::= '0' | ... | '9'

B.2 Description of Configuration Entities

A configuration description (input for code_gen) has the following format:

<*conf_description*> ::= Configuration class <*name*>
{< *required_method* >}* <*interface*>
{< *channel_desc* >}+

<*required_method*> ::= Requires <*method*>
{',' < *method* >}* ';'

<*method*> ::= <*name*> <*name*> <*arg_txt*>

<*interface*> ::= Interface <*name*> ';'

<*channel_desc*> ::= Channel <*name*>
{< *ch_spec* >}+ Channel_End

<*ch_desc*> ::= Reuse <*value*> ';'
Send proc_call ';'
Object_Wait <*value*> ';'

Execute_Method <*value*> ';'

Object_Send_Ack <*value*> ';'

Accepts Attributes

<*name*> {',' < *name* >}* ';'

Filename <*filename*> ';'

Function_Name <*name*> ';'

<*value*> ::= Yes | No

<*arg_txt*> ::= '('{< *letter* > | < *number* > | ' ' | ','}*')'

<*filename*> ::= <*name*>'.c'

<*name*> ::= <*letter*> {< *letter* > | < *digit* > | '_' }*

<*letter*> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'

<*number*> ::= {< *digit* >}+

<*digit*> ::= '0' | ... | '9'

B.3 Description of Application Components

Application descriptions are defined as follows;

<*application*> ::= Application <*name*> ';'

{< *spec* >}*

<*spec*> ::= Interface <*interface_list*>

| Configuration <*conf_list*>

| Sourcefile <*source_list*>

| EXTRA_INC_DIR <*include_list*>

| EXTRA_OB_FILE <*ob_list*>

| EXTRA_LIBRARY <*lib_list*>

| EXTRA_LIB_PATH <*path_list*>

| EXECUTABLE_NAME <*name*>

<*interface_list*> ::= <*interface*> {',' < *interface* >}*

<*interface*> ::= <*name*> { IMPLEMENTATION < *filename* > }?

{< *path* >}?

<*path*> ::= PATH '"' <*name*> '"'

<*conf_list*> ::= <*name*> {',' < *name* >}*

<*source_list*> ::= <*filename*> {',' < *filename* >}*

<*include_list*> ::= <*name*> {',' < *name* >}*

<*ob_list*> ::= <*name*> {',' < *name* >}*

<*lib_list*> ::= <*name*> {',' < *name* >}*

<*path_list*> ::= <*name*> {',' < *name* >}*

<*filename*> ::= <*name*>'.c'

<*name*> ::= <*letter*> {< *letter* > | < *digit* > | '_' }*

<*letter*> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'

<*digit*> ::= '0' | ... | '9'

Bibliography

[AHP94] Brent Agnew, Christine Hofmeister, and James 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.

[AS95] Mustaque Ahamad and Karsten Schwan. The COBS Project. http://www.cc.gatech.edu/systems/projects/COBS, 1995.

[Ast96] Hernan Astudillo. *Evaluation and Realization of Modeling Alternatives: Supporting Derivation and Enhancement.* PhD thesis, College of Computing, Georgia Institute of Technology, April 1996.

[Ban96] Arindam Banerji. *Protected Shared Libraries — Modularity and Extensibility for Commercial Operating Systems.* PhD thesis, Department of Computer Science and Engineering, University of Notre Dame, August 1996.

[Be85] Kenneth P. Birman and et.al. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, pages 502–508, June 1985.

[Bec94] Thomas Becker. Application-transparent fault tolerance in distributed systems. In *Proc. of the Second International Workshop in Configurable Distributed Systems.* IEEE Computer Society Press, May 1994.

[BKC94] A. Banerji, D. Kulkarni, and D. Cohn. A framework for building extensible class libraries. In *Proc. of the USENIX C++ Conference*, pages 26–41, 1994.

[BKTC93] A. Banerji, D. Kulkarni, J. Tracey, and D. Cohn. The substrate model and architecture. In *Proc. of the 3rd International Workshop on Object-Orientation in Operating Systems*, pages 31–41. IEEE, 1993.

[BS91] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[CAK+96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 108–115. IEEE Computer Society Press, May 1996.

[CB93] J. Bradley Chen and Brian 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.

[CCLP81] George Cox, William M. Corwin, Konrad K. Lai, and Fred J. Pollack. A unified model and implementation for interprocess communication in a multiprocessor environment. In *Proc. of the 8th ACM Symposium on Operating System Principles*, pages 44–53, Dec. 1981.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical report, Computer Science, Carnegie-Mellon University, CMU-CS-88-154, June 1988.

[CDL+93] E.M. Chaves Jr., P.C. Das, T.L. LeBlanc, B.D. Marsh, and M.L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–192, May 1993.

[CDS96] *Proceedings of the 3rd International Conference on Configurable Distributed Systems.* IEEE Computer Society Press, May 1996.

[CFH+93] John B. Carter, Bryan Fork, Mike Hibler, Ravindra Kuramkote, Jeffrey Law, Jay Lepreau, Douglas B. Orr, Leigh Stoller, and Mark Swanson. FLEX: a tool for building efficient and flexible systems. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 198–202, Napa, CA, October 1993.

[CK93] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming.* MIT Press, 1993.

[CMS93] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on large-scale multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-93-25, May 1993. Also in in 'Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS-4)', Sept. 1993.

[CMS96] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on multiprocessors. *IEEE Transactions on Software Engineering*, 22(2):132–152, February 1996.

[CPW93] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1993.

[CRJ87] Roy Campbell, Vincent Russo, and Gary Johnson. Choices (class hierarchical open interface for custom embedded systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[DF94] Scott Danforth and Ira Forman. Reflections on metaclass programming in SOM. In *Proc. of OOPSLA '94*, pages 440–452. ACM Press, October 1994.

[DPH92] Peter Druschel, Larry Peterson, and Norman Hutchinson. Beyond microkernel design: Decoupling modularity and protection in Lipto. In *Proc. of the 12th Internacional Conference on Distributed Computing Systems*, pages 512–520, June 1992.

[Dru93] Peter Druschel. Efficient support for incremental customization of OS services. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 186–190, December 1993.

[ea94] G. Kiczales et al. Open implementations: A metaobject protocol approach. In *Proc. of the 9th Conference on Object-Oriented Programming Systems, Language, and Applications*, 1994. Tutorial notes.

[ED96] Markus Endler and Anil J. D'Souza. Supporting distributed application management in Sampa. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 177–184, May 1996.

[Eis94] Greg Eisenhauer. Portable self-describin binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, Atlanta, GA, 1994.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole 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.

[ES96] Greg Eisenhauer and Karsten Schwan. Parallelization of a molecular dynamics code. *Journal of Parallel and Distributed Computing (SPDT)*, 34(2), May 1996.

[ESS96] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. From interactive high performance programs to distributed laboratories: A research agenda. In *Proc. of the SPDP'96 Workshop on Program Visualization and Instrumentation*, October 1996.

[FBYR88] Alessandro Forin, Joseph Barrera, Michael Yound, and Richard Rashid. Design, implementation and performance evaluation of a distributed shared memory server for Mach. Technical Report CMU-CS-88-1, CMU, 1988. A short version appears in Proc. Winter 1989 Winter Conference.

[FBZ94] 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.

[FDM94] Ira Froman, Scott Danforth, and Hari Madduri. Composition of before/after metaclasses in SOM. In *Proc. of OOPSLA'94*, pages 427–439. ACM Press, October 1994.

[FJL+88] Geoffrey C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors.* Prentice-Hall, 1988.

[For96] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, 1996.

[GEK+95] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proc. of FRONTIERS'95*, February 1995. Also available as Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology.

[GGSW88] Ahmed Gheith, Prabha Gopinath, Karsten Schwan, and Peter Wiley. Chaos and chaos-art: Extensions to an object-based kernel. In *IEEE Computer Society Fifth Workshop on Real-Time Operating Systems, Washington, D.C.* IEEE, April 1988.

[Ghe90] Ahmed Gheith. *Support for Multi-Weight Objects, Invocations, and Atomicity in Real-Time Systems.* PhD thesis, Georgia Institute of Technology, College of Computing, Aug. 1990.

[GMSS94] Ahmed Gheith, Bodhi Mukherjee, Dilma Menezes da Silva, and Karsten Schwan. Configurable objects and invocations. In *Proc. of the Second International Workshop in Configurable Distributed Systems*, pages 92–104. IEEE Computer Society Press, may 1994.

[GS89] Ahmed Gheith and Karsten Schwan. Chaos-art: Kernel support for atomic transactions in real-time applications. In *Nineteenth International Symposium on Fault-Tolerant Computing, Chicago, IL*, pages 462–469, June 1989. Also see GIT-ICS-90/06, College of Computing, Georgia Tech, Atlanta, GA 30332.

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

[GVS94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29(9):140–148, Sept. 1994.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[HP91] 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.

[HPM93] Graham Hamilton, Michael Powell, and James 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.

[HvDvS+95] Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An object model for flexible distributed systems. In *Proc. of ASCI'95, ASCI T.U. Delft*, pages 69–78, 1995.

[IRS96] Daniela Ivan-Rosu and Karsten 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.

[JMY+96] Rakesh Jha, Mustafa Muhammad, Sudharkar Yalamanchili, Karsten Schwan, Daniela Ivan-Rosu, and Chris DeCastro. Adaptive resource allocation for embedded parallel applications. In *Third International Conference on High Performance Computing*. IEEE, December 1996.

[Jon93] 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.

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

[KL93] Gregor Kiczales and John Lamping. Operating systems: Why object oriented? In *International Workshop in Object Oriented Operating Systems*, pages 25–30, December 1993.

[KM85] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[KMF90] Jeff Kramer, Jeff Magee, and Anthony Finkelstein. A constructive approach to the design of distributed systems. In *Proc. of the 10th International Conference on Distributed Computing Systems*, pages 580–587, May 1990.

[KN93] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in Spring. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–13. ACM PRESS, Dec 1993.

[KSP94] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive parallelism under Equus. In *Proc. of the 2nd International Workshop in Configurable Distributed Systems*, pages 172–182. IEEE Computer Society Press, May 1994.

[KSS+96] Thomas Kindler, Karsten Schwan, Dilma M. Silva, Mary Trauner, and Fred Alyea. Parallelization of spectral models for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[LCC+75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proc. of the 5th ACM Symposium on Operating System Principles*, Nov. 1975.

[LG85] I. Lee and V. Gehlot. Language constructs for distributed real-time programming. In *Proc. of the 6th Real-Time Systems Symposium, San Diego, CA*, pages 57–66. IEEE, Dec. 1985.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of ACM*, 31(3):300–313, March 1988.

[LKAS93] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. An architecture and toolkit for parallel and configurable protocols. In *Proceedings of the International Conference on Network Protocols (ICNP-93)*, September 1993.

[LM94] M. Little and D. McCue. The replica management system: a scheme for flexible and dynamic replication. In *Proc. of the Second International Workshop in Configurable Distributed Systems.* IEEE Computer Society Press, May 1994.

[LS96] Mark C. Little and Santosh K. Shrivastava. Using application specific knowledge for configuring object replicas. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 169–176, May 1996.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proc. of OOPSLA '87*, pages 147–155. ACM Press, October 1987.

[Mas92] H. Massalin. *Synthesis: An Efficient Implementation of Operational System Services.* PhD thesis, Columbia University, 1992.

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. A constructive development environment for parallel and distributed programs. In *Proc. of the Second International Workshop in Configurable Distributed Systems*, pages 4–14. IEEE Computer Society Press, May 1994.

[MK83] J. Magee and J. Kramer. Dynamic configuration for distributed real-time systems. In *Proc. of the International Conference on Parallel Processing*, pages 277–288, Aug. 1983.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, Dec. 1989.

[MRS+90] 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.

[MS93] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 59–66, July 1993.

[MSSG94] Bodhisattwa Mukherjee, Dilma Menezes da Silva, Karsten Schwan, and Ahmed Gheith. KTK: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*, 1:259–270, 1994.

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

[MW91] Keith Marzullo and Mark Wood. Making real-time systems reactive. *ACM Operating Systems Review*, 25(1), January 1991.

[NHK93] Michael N. Nelson, Graham Hamilton, and Yousef A. Khalidi. Caching in an object oriented system. In *International Workshop in Object Oriented Operating Systems*, pages 95–106, December 1993.

[Obj] Object Management Group. The OMG web site. http://www.omg.org.

[Obj95] Object Management Group. The Common Object Request Broker: Architecture and Specification. Available from the OMG Web site (http://www.omg.org), July 1995.

[PAB95] Calton Pu, Tito Autrey, and Andrew Black. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, Dec 1995.

[Pae93] Andrews Paepcke, editor. *Object-Oriented Programming - The CLOS Perspective.* MIT Press, 1993.

[PB96] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proc. of the 2nd Symposium on Operating System Design and Implementation (OSDI)*, October 1996.

[PHOA89] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abbot. RPC in the x-kernel. In *Twelfth ACM Symposium on Operating Systems*, pages 91–101. ACM, Dec. 1989.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[Rei91] Gerhard Reinelt. TSPLIB — a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[RJea89] Richard Rashid, Danile Julin, and et al. Mach: A system software kernel. In *Proc. of the 34th IEEE Computer society International Conference (COMPCON 89)*, pages 176–178, February 1989.

[SB90] Karsten Schwan and Win Bo. Topologies – distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990. Older version available as OSU-CISRC-3/88-TR11.

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

[SBWT85] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. Gem: Operating system primitives for robots and real-time control. In *Proc. of the International Conference on Robotics and Automation, St. Louis, Missouri*, pages 807–813. IEEE, March 1985. Also published as article in ACM TOCS.

[Sch] Karsten Schwan et al. The falcon monitoring and steering system. http://www.cc.gatech.edu/systems/projects/FALCON/.

[Sch83] Karsten Schwan. Poster session: The Issos project. In *Proc. of the 9th Symposium on Operating Systems Principles.* Assoc. Comput. Mach., Oct. 1983.

[Sch93] D. Schmidt. The adaptive communication environment. In *Proc. of the 11th Sun User Group Conference*, 1993.

[Sch94] Douglas Schmidt. An object-oriented framework for dynamically configuring extensible distributed systems. *IEE Distributed Systems Engineering Journal*, 2(4), Dec 1994.

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of Arjuna: a programming system for reliable computing. *IEEE Software*, 8(1):63–73, January 1991.

[SFG+91] Karsten Schwan, Harold Forbes, Ahmed Gheith, Bodhisattwa Mukherjee, and Yiannis Samiotakis. A Cthread library for multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, GIT-ICS-91/02, Jan. 1991.

[SG96] Bala Swaminathan and Kenneth J. Goldman. Data handles and virtual connections. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 19–26, May 1996.

[SGZ90a] K. Schwan, A. Gheith, and H. Zhou. Chaos-arc: A kernel for predictable programs in dynamic real-time systems. In *7th IEEE Workshop on Real-Time Operating Systems and Software, Univ. of Virginia, Charlottesville*, pages 11–19, May 1990.

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

[Sha86] 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.

[Sha94] David L. Shang. Dynamic parametrism: An efficient way to present the flexibility of dynamic systems. In *Workshop on Flexible Systems, OOPSLA '94*, 1994.

[Sha95] Marc Shapiro. Matching operating systems to application needs. *Operating Systems Review*, 29(1):47–51, January 1995.

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley & Sons, Inc., 1996.

[SJ86] Karsten Schwan and Anita K. Jones. Flexible software development for multiple computer systems. *IEEE Transactions on Software Engineering*, SE-12(3):385–401, March 1986.

[Smi82] B. C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, Laboratory for Computer Science, MIT, 1982.

[Som92] Ian Sommervile. *Software Engineering.* Addison-Wesley Pub Co, 4th edition, 1992.

[SR84] Karsten Schwan and Rajiv Ramnath. Adaptable operating software for manufacturing systems and robots: A computer science research agenda. In *Proc. of the 5th IEEE Real-Time Systems Symposium*, pages 255–262, Dec. 1984.

[SR87] J. Stankovic and K. Ramamritham. The design of the Spring kernel. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 146–157, Dec. 1987.

[SS97] Dilma Menezes da Silva and Karsten Schwan. CTK: configurable object abstractions for multiprocessors. Technical Report GIT-CC-97-03, Georgia Institute of Technology, Atlanta, GA 30332, January 1997. Submitted to IEEE Transactions on Software Engineering.

[TG89] 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.

[vDHT95] Leendert van Doorn, Philip Homburg, and Andrew S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proc. of the IEEE Workshop on Hot Topics In Operating Systems*, 1995.

[VH96] Alistair C. Veich and Normal C. Hutchinson. Kea – a dynamically extensible and configurable operating system kernel. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 236–242, May 1996.

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

[WS96] Ian Warren and Ian Sommerville. A model for dynamic configuration which preserver applicatin integrity. In *Proc. of the 3rd International Conference on Configurable Distributed Systems*, pages 81–88, May 1996.

[Yok92] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proc. of OOPSLA'92*, pages 414–434. ACM Press, October 1992.

[ZC95] Chris Zimmermann and Vinny Cahill. Open to sugestions: on adaptable, distributed application support architectures. In *European Reserach Seminar on Advances in Distributed Systems*, 1995.

[ZSA92] Hongyi Zhou, Karsten Schwan, and Ian Akyildiz. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. In *Proc. of the 1992 Real-time Systems Symposium, Phoenix, AZ.* IEEE, Dec. 1992. Also available as technical report GIT-CC-91/40.

Vita

Dilma Menezes da Silva was born in June 16th 1965, although from the very beginning the documents papers showed 6/25/65 as her birth date (and still do). The mistake has been very useful so far: she has multiple birthday parties in June, she pretends to be a few days younger, and she chooses her astrology sign from two options, according to which prediction she likes best for the day.

She was born in São Paulo, Brazil. Her family migrated from the Northeast, and she still has many relatives in Bahia, a very beautiful and culturally rich state.

Dr. Silva enjoyed going to school since her first days in kindergarten. She went to a private kindergarten and looked very cute in the uniform (which involved a

mini-skirt and a batman-like cape). She went to public school for the rest of her education, meeting some good literature and math teachers. She feels outraged by the deterioration in the public elementary education system, and hopeless because she can not see how to act upon this problem.

She was admitted to the University of São Paulo in 1983. She planned to major in Math, but in the end of her first year she was decided to get a Bachelor degree in Computer Science, what she did in 1986. She felt very enthusiastic about most of the courses she took. Going to graduate school was the natural path, since she guessed that Cobol programming could get boring very quickly. She was very fortunate in her choices for advisor (Dr. Arnaldo Mandel) and dissertation topic (object oriented programming). She got her Master's in Computer Science from University of São Paulo in 1990. She was hired as a lecturer in 1987, and discovered that the combination of learning and teaching made for a very exciting career. In the same week she presented her Master's degree dissertation, flew to the United States and joined the PhD program in Computer Science at Georgia Tech.

Dr. Silva arrived in Atlanta in September, 1990. She was overwhelmed about all the things she had to learn about software systems, and even felt a little bitter about having studied so much theory "for nothing". She joined Professor Karsten Schwan's research group, and was fortunate in working with many bright and helpful people. Most important of all, she met very good friends that made her life in Georgia Tech

very joyful. She presented her oral PhD Defense in August 23rd, 1996.

Dr. Silva is currently a faculty member in University of São Paulo, São Paulo, Brazil. She likes movies and books *a lot.*