TORNADO: MAXIMIZING LOCALITY AND CONCURRENCY IN A SHARED-MEMORY MULTIPROCESSOR OPERATING SYSTEM

by

Benjamin Gamsa

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Tornado: Maximizing Locality and Concurrency
in a Shared-Memory Multiprocessor Operating System

Benjamin Gamsa
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1999

This dissertation presents novel operating system structuring techniques for dealing with the problems of scalability in shared-memory multiprocessors. By using an object-oriented structure, with each virtual and physical resource represented by an independent object, Tornado eliminates most shared global objects, thus reducing contention and increasing locality. To improve performance for contended components, Tornado uses a new structuring technique called *Clustered Objects* that allows an object to be partitioned and distributed across the machine in a manner transparent to the outside consumers of the object. In addition, Tornado includes a new interprocess communication facility, called the *Protected Procedure Call* facility, that provides the locality and concurrency required to allow microkernels to scale effectively on multiprocessors. This dissertation also explores some of the other issues in multiprocessor operating system design, such as efficient lock and memory allocation implementations, as well as the interactions between concurrency control and object destruction.

A prototype implementation of the techniques described have been implemented as part of the Tornado operating system for the NUMAchine multiprocessor. This dissertation explores both the design aspects of the system as well as experiences gained through its implementation and use on both NUMAchine and a complete machine simulator, SimOS.

Preface

Like most large systems projects, Tornado is a group effort. The contributions of many people made the system possible.

- Orran Krieger: implemented the original File System (HFS) and user-level I/O library (AFS), and helped flesh out the Tornado architecture through many fruitful discussions.
- Eric Parsons: implemented the locking routines and name service.
- Paul Lu: implemented the debugging stub and event service.
- Karen Reid: ported NFS to Tornado.
- Daniel Wilks: implemented the memory manager.
- Jonathan Appavoo: refined the implementation of the Clustered Object system.
- Derek Devries: ported the pipe server, HFS, and other utilities to Tornado.

My primary role was as systems architect and lead implementor. In that position, I was responsible for the overall system design, including the object-oriented structure of the system, the clustered object design, the memory management design, the process management design, and the IPC design. I was also responsible for most of the implementation of these and other components.

Contents

1 Introduction **1**

1.1 Problem Description 1

1.2 System design goals 3

1.3 Previous Approaches 4

1.4 Design Approach 7

1.5 Dissertation outline 8

2 Background **10**

2.1 Multiprocessor Hardware Issues 10

2.1.1 Physical parallelism 11

2.1.2 Cache coherence 12

2.1.3 Cache miss latency 13

2.1.4 Summary 14

2.2 Design principles 15

2.2.1 Structuring data for caches 15

2.2.2 Locking data 16

2.2.3 Localizing data accesses 17

2.3 Summary 18

3 System Overview **20**

3.1 Operating environment 20

3.2 Experimental Setup 22

3.2.1 NUMAchine 22

3.2.2 SimOS simulator 23

3.3 Tornado Architecture 24

3.3.1 Operating System structure 24

3.3.2 Tornado programmer interface 26

3.3.3 Basic programmer-visible abstractions . . . 27
3.3.4 Key kernel classes . . . 27
3.3.5 Kernel and system server design strategies . . . 31
3.4 Summary . . . 37

4 Clustered Objects **38**
4.1 Motivation . . . 38
4.2 Implementation . . . 41
4.2.1 Object References and Translation . . . 44
4.2.2 Translation Misses . . . 45
4.2.3 Managing the translation table memory . . . 48
4.2.4 Destruction . . . 48
4.2.5 External Access to Clustered Objects . . . 53
4.3 Examples . . . 56
4.4 Performance evaluation . . . 58
4.4.1 Component measurements . . . 58
4.4.2 Sample clustered objects . . . 60
4.5 Open Issues . . . 67
4.6 Related work . . . 68
4.7 Summary . . . 69

5 Protected Procedure Call **70**
5.1 Motivation . . . 70
5.2 PPC model . . . 71
5.2.1 Overview of PPC . . . 72
5.2.2 Parameter passing mechanisms . . . 73
5.2.3 Naming and authentication . . . 74
5.2.4 Extended application of PPCs . . . 75
5.2.5 Remote PPC . . . 76
5.2.6 Error handling . . . 76
5.2.7 Process Initialization . . . 77
5.3 Implementation . . . 77
5.3.1 Overview . . . 77
5.3.2 Process Descriptor . . . 79
5.3.3 PortAnnex table . . . 79
5.3.4 Stacks . . . 80

5.3.5 IN/OUT pages . . . 82
5.3.6 Scheduling . . . 82
5.3.7 Exceptional conditions . . . 82
5.3.8 Hardware exceptions . . . 83
5.3.9 Remote PPCs . . . 84
5.4 Performance . . . 86
5.5 Open Issues . . . 90
5.6 Related work . . . 90
5.7 Summary . . . 91

6 Support Infrastructure **92**
6.1 Locking facility . . . 93
6.1.1 Locking issues . . . 93
6.1.2 Tornado locking facility . . . 96
6.1.3 Performance evaluation . . . 100
6.1.4 Open Issues . . . 101
6.1.5 Related Work . . . 102
6.2 Memory allocation subsystem . . . 103
6.2.1 Multiprocessor allocator issues . . . 103
6.2.2 Tornado allocation facility . . . 107
6.2.3 Performance evaluation . . . 115
6.2.4 Open issues . . . 117
6.3 Related Work . . . 117
6.4 Summary . . . 118

7 Overall System Performance **119**

8 Overall Summary **123**
8.1 Object-Oriented Structure . . . 123
8.2 Clustered Objects . . . 124
8.3 Clustered Object Garbage Collection . . . 126
8.4 Protected Procedure Calls . . . 127
8.5 Locks . . . 128
8.6 Memory Allocation . . . 129
8.7 Overall Open Issues . . . 130

9 Lessons Learned **131**

10 Concluding remarks **134**

Bibliography **135**

Chapter 1

Introduction

With processor speeds increasing faster than memory or multiprocessor interconnects, designing a shared-memory multiprocessor operating system with good performance is becoming increasingly difficult. Both scaling the operating system up to the size of the larger multiprocessors and providing performance competitive with uniprocessor systems is a challenging task.

This dissertation attempts to address these issues by presenting new system software structuring techniques and key support facilities that help improve locality and concurrency. More specifically, we present an object-oriented design that increases locality and concurrency by encapsulating each resource instance within an independent object. We further extend this with a novel structuring technique called *Clustered Objects* that improves scalability by supporting the replication, migration, and partitioning of shared data structures to increase locality and reduce contention. We also present a new interprocess communication facility, called PPC, that provides a highly efficient and concurrent glue for the multiple servers of the microkernel system. Finally, we describe locking and dynamic memory allocation facilities that aid efficient system design by reducing the costs of these key components and improving their scalability.

This work is presented within the context of a fully functional multiprocessor operating system, Tornado, running on an experimental shared-memory multiprocessor called NUMAchine. The dissertation consists of the design, implementation, and evaluation of Tornado.

The rest of this chapter provides an overview of the issues to be addressed, our design goals, previous approaches to the problem, and a general outline of the direction we took.

1.1 Problem Description

Designing a high performance operating system for shared-memory multiprocessors is fundamentally more difficult than designing one for uniprocessors. Multiprocessors require that a distinct set of issues be considered. For small-scale bus-based shared-memory multiprocessors (4–16 processors), the overhead of cache consistency requires careful attention to the organization and sharing of data in order to reduce the number of cache misses. For example, the cost of simple list manipulations can increase by over two-orders of magnitude when the list is shared by just two processors. In addition, more careful attention must be given to the concurrency control architecture to ensure correctness and efficiency. Contention for locks and shared data must be carefully monitored for potential bottlenecks.

With larger systems (32 processors and up), there has been a tendency to use what is called a Non-Uniform Memory Access time (NUMA) architecture, in which processors, memory, and I/O devices are physically distributed and interconnected by a general interconnection network, such as a mesh, ring, or torus. Although a NUMA architecture increases total aggregate bandwidth, it also introduces higher latencies for remote memory accesses and the potential for increased contention within the network. In addition, the size of the system (independent of architecture) increases the probability of contention at any system component, such as at the memory or I/O modules. As a result of all of these attributes, attention must be paid to the placement of data in order to maximize locality and reduce hot spots. Although cache coherence partially addresses locality issues by automatically migrating and replicating data, it is well known that operating systems have very poor cache hit rates and therefore relying strictly on the caches may result in dramatically poorer performance than what is possible.

Thus, to achieve good performance, it may be necessary to apply optimizations such as data alignment, padding, regrouping, replication, migration, or sometimes completely different data structures and algorithms altogether. Without some support, the systems programmer will need to be aware of all the intricacies of the multiprocessor hardware and explicitly optimize for them.

Unfortunately, structuring the operating system so as to maximize the performance of large-scale applications can have a detrimental affect on small-scale and sequential applications. For example, to reduce a hot spot caused by concurrent page-faults for a common region of memory, the key kernel data structures may be randomly distributed across the system. As a side effect, however, this may eliminate the possible localization of those same data structures when no sharing is occurring, leading to higher overheads. In addition, elaborate data structures and fine-grained locking help increase concurrency but may result in unacceptable and unnecessary overhead, especially when the operating system is deployed on small-scale systems. Hence it is important for the operating system to effectively scale up and down, both with respect to the hardware and the application workload.

Multiprocessor workloads can pose many difficult challenges for the operating system, especially for large scale systems. One of the key benefits of large-scale shared-memory multiprocessors over message-passing multiprocessors is the ability to use them as both a general purpose compute server and as a target for high-performance large-scale parallel applications. The operating system must thus support multiprogramming and simultaneously provide high responsiveness for what will likely be a large number of small sequential interactive applications, and efficient resource management for a smaller number of large parallel non-interactive applications.

Large-scale parallel applications have a number of special requirements. These include: the desire to control their own resources (fueled by the desire to "get the operating system out of the way"), predictable resource allocation to support compiler and run-time system optimizations that are resource dependent (such as how much memory is available), interactive response for visualization or debugging, and repeatability and consistency for performance debugging. And, of course, the operating system must balance these needs against the demands of all other applications.

In summary, given these requirements, the primary objective for Tornado is to achieve high throughput and responsiveness for both large and small applications on both large and small-scale multiprocessors. The structure of the operating system must fit both the scale and distributed nature of the hardware and the pattern of requests expected from the workload. To achieve this, the operating system must be designed to avoid contention in the operating system data structures, increase locality and reduce cache coherence overhead by reducing true and false sharing, and provide policies that match the needs of the workload. At the same time, while attempting to achieve good performance for large systems and applications, it is important that the operating system not hurt either small application or small system performance. At the moment, no existing operating system can meet all of these goals.

1.2 System design goals

Given the issues discussed above, Tornado's primary design goal is the following:

> *The per-application operating system overhead should depend only on the number of resources used by the application, not on the size of the system overall.*

That is, an application under Tornado should execute with the same efficiency on a large-scale multiprocessor as it would on a small-scale multiprocessor.

In addition, Tornado has a number of secondary design goals:

- The system should degrade gracefully as contention for a single resource grows, whether the contention comes from a single large-scale parallel application or a large-number of sequential applications;

- The performance of a given workload for a given system size under Tornado should be "comparable" to that of a system optimized just for the given workload and hardware (although we are willing to accept a minimal cost to obtain the scalability benefits);
- The system should provide multiple policies and implementations so as to match the requirements of a diverse set of application workloads;
- The overall complexity of the system must remain manageable and permit a natural evolution as the workload and hardware changes.

1.3 Previous Approaches

There have been three main approaches to developing operating systems for scalable multiprocessors. The first approach is to take a uniprocessor operating system and make all the changes necessary to make it function correctly on a multiprocessor, and then tweak it incrementally until it runs efficiently on whatever size machine it is being targeted for. This has been the approach taken by most mainstream UNIX multiprocessor vendors, and has allowed them to slowly increase the scalability of their systems from a couple of processors up to about 16 or 32 today [Chapin *et al.*, 1995a, Talbot, 1995, McCrocklin, 1995, Campbell *et al.*, 1991a, Presotto, 1990]. However, this tends to be a slow, ad hoc, and laborious task, that has to be repeated with each data structure every time the system needs to be expanded, and still results in a system that falls short of being able to take full advantage of the hardware when stressed. This is exemplified in Figure 1.1 that shows the results of a few simple micro-benchmarks run on a number of commercial multiprocessor operating systems as well as on Tornado.[1] (More details on these experiments are provided in Chapter 7.) For each commercial operating system considered, there is a significant slowdown when simple operations are issued in parallel that should be serviceable completely independently of each other. We experienced similar difficulties in a multiprocessor operating system we developed previously ourselves [Unrau *et al.*, 1995].

We conjecture that these systems perform inadequately for two reasons. First, as suggested above, it appears that they were developed in an evolutionary way, with locking protocols and internal structures adapted from their earlier uniprocessor counterparts and subsequently tuned for improved performance. Second, in the tuning of these systems, the focus was almost entirely on increasing concurrency, primarily by repeatedly breaking contended locks into finer grained locks. We contend that this approach is fundamentally problematic. Adding fine-grained locking to a

[1] While micro-benchmarks are not necessarily a good measure of overall performance, these results do show that the existing systems can have performance problems.

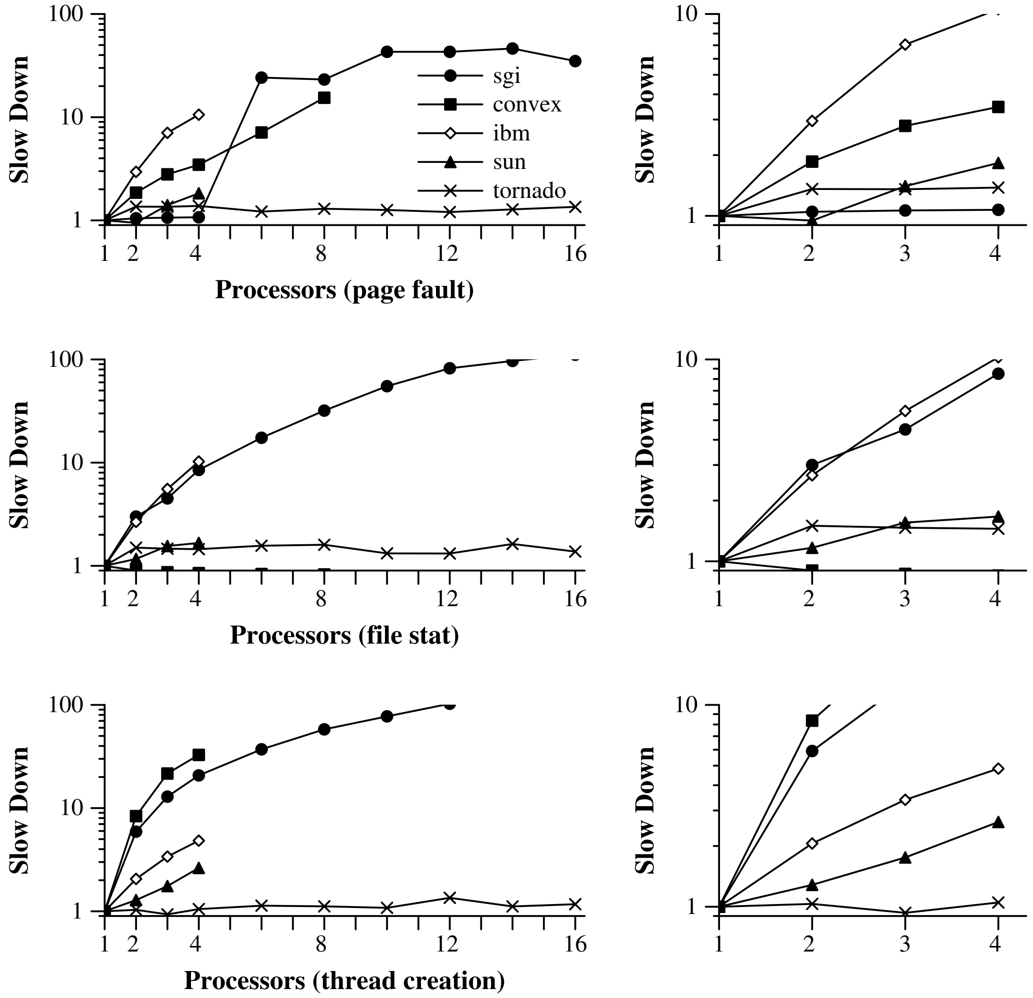


Figure 1.1: *These graphs show normalized execution time for three simple micro-benchmarks across a number of different systems. The slowdown is shown on a log scale. Each benchmark is shown for up to 16 processors (on the left) with a blowup up to 4 processors (on the right). Details of these tests are presented later in Chapter 7.*

locking protocol developed for uniprocessors adds overhead to the critical paths and adds substantial complexity to the system, making the software difficult to maintain and optimize. Moreover, uniprocessor data structures have little or no locality, so the operating system suffers the overhead of many communication cache misses. In our experience, it is just as important to optimize for good cache locality with minimal true and false sharing as it is to optimize for good concurrency. Poor locality can lead to substantial miss overhead that will be exacerbated in the next generations of multiprocessors, where a communication miss is likely to incur a latency of hundreds of processor cycles.

Another approach to developing scalable multiprocessor operating systems is to start with a distributed operating system which is inherently scalable, and then provide the single system image features expected of a single multiprocessor operating system [Popek and Walker, 1985, Zajcew *et al.*, 1993, Ousterhout *et al.*, 1988]. The problem with this approach is that the protocols tend to be heavy-weight since they are designed for an environment in which communication is expensive, and it is exceedingly difficult to provide true single system semantics on top of a distributed software architecture. Most of the effort for this work has focus on distributed memory multiprocessors where traditional multiprocessor operating systems techniques (such as those above) are not applicable. However, even in this limited domain they have not been very successful at providing both high performance and a single-system image.

A final approach, taken by Hurricane [Unrau *et al.*, 1995], the predecessor to Tornado, as well as the Stanford Hive project [Chapin *et al.*, 1995b], is to take an in-between approach and start with a small-scale multiprocessor operating system and then extend it to larger systems by applying distributed system protocols to connect multiple instances of the operating system within a single machine.[2] This structuring technique, called *hierarchical clustering* in Hurricane, manages physically proximate resources in a tightly coupled fashion, and more distantly-connected resources in a gradually more loosely-coupled fashion. The idea is to provide the high performance of the tight coupling of a small-scale multiprocessor for most local interactions, while providing the scalability of a distributed system.

Our experience with this latter approach in Hurricane showed that it can deliver the desired scalable multiprocessor performance [Unrau *et al.*, 1995], but that it also has the disadvantages of both previous approaches (scaling up a small-scale operating system and increasing the coupling of a distributed system). First, all the multiprocessor issues of concurrency control and performance need to be addressed (albeit for a smaller system), and all the single system image issues of a distributed system must also be solved. Second, a fixed cluster size is applied to all resources, sometimes imposing a cluster size that is either too large or too small for a given resource for maximum performance. Worse, resources that do not need to be clustered at all for performance still must pay the complexity cost of clustering to function correctly in the distributed environment. Further, as it was designed, all components of the system were required to share the same clustering hierarchy and knowledge of this hierarchy was embedded in all parts of the system. As a result, hierarchical clustering failed to provide the transparency and flexibility necessary for a maintainable system. Finally, this approach greatly complicates the development of the system. From a pragmatic point of view, as development proceeds, one tends to have either a small scale system that is reliable, or a large scale system that is unstable, but nothing in between, making it difficult to debug and develop clustering solutions for different components without the clustering problems of the other components getting in the way.

[2] Note that the primary motivation for this approach in Hive was for fault containment.

As a result of these problems with hierarchical clustering, both groups have moved on to other approaches. In the case of the Hive group, recent effort has focussed on a new approach, called Disco [Bugnion *et al.*, 1997], that provides a scalable virtual machine on which multiple traditional operating systems can be run, each using some subset of the available processors. This provides a transitional approach to scalability: the size of the partition given to each operating system instance can grow over time as these systems are tuned for greater scalability. However, this partitioning of resources defeats the purpose of having a single large-scale system. In the case of Hurricane, the new approach is Tornado, which is the focus of this dissertation.

1.4 Design Approach

In Tornado, all system components were designed from scratch with the primary overriding design principle of mapping any locality and independence that might exist in operating system requests from applications, to locality and independence in the servicing of these requests in the operating systems and system servers. We found that we could apply this principle by using a small number of relatively simple techniques in a systematic fashion. As a result, Tornado has a simpler structure than other multiprocessor operating systems, and hence can be more easily maintained and optimized.

More specifically, the design of Tornado is based on the observation that: (i) operating systems are driven by the requests of applications on virtual resources, (ii) to achieve good performance on multiprocessors, request to different resources should be handled independently, that is, without accessing any common data structures and without acquiring any common locks, and (iii) the requests should, in the common case, be serviced on the same processor they are issued on. We achieve this in Tornado by adopting an object-oriented approach:

> *Each virtual and physical resource in the system is represented by an independent object so that accesses on different processors to different objects remain independent.*

However, in the case where a single virtual resource is the target of multiple concurrent requests, a single object is insufficient for scalability. What is desired is an approach, similar to hierarchical clustering, that partitions and distributes a resource across the system, but applied on an object-by-object basis rather to an entire small-scale operating system. The resulting system, which we term *Clustered Objects*, allows an object to be partitioned into multiple *representatives* with independent requests on different processors handled by different representatives of the object. Thus, simultaneous requests from a parallel application to a single virtual resource (i.e., page faults to different pages of the same memory region) can be handled efficiently with as much locality and concurrency as possible.

Although a clustered object is composed of multiple representative objects—allowing the state of the single logical object to be replicated, migrated, or distributed as best fits the particular resource it is managing—it still provides the abstraction of a single object to the rest of the system. Thus, clustered objects provide the locality-enhancing benefits of hierarchical clustering but do so within an object-oriented framework, with its implementation-hiding benefits. This provides more freedom to develop individual structural solutions for different components of the system independently, making the best use of the traits of a shared-memory multiprocessor—efficient and convenient fine-grained communication—and the best use of distributed operating system techniques—which help reduce contention by distributing system components and reducing sharing.

Although clustered objects help address structural issues within the kernel, an operating system is often composed of many cooperating servers, particularly in the case of a microkernel such as Tornado. It is therefore important to maintain the locality and concurrency of the clustered object design even in the communication between the kernel and system servers. Tornado therefore depends on a highly concurrent and localizing interprocessor communication (IPC) subsystem. The Tornado *Protected Procedure Call* facility was designed to preserve the locality and concurrency in client requests, and yet perform competitively with the best uniprocessor IPC facilities. This allows repeated requests to the same object to be serviced on the same processor as the client thread, while concurrent requests are automatically serviced by different server threads without any need for synchronization to start the server threads.

Finally, an infrastructure is required that provides the locality, concurrency, and efficiency needed by each component as part of such basic facilities as dynamic memory allocation, concurrency control, and performance monitoring.

1.5 Dissertation outline

The rest of this dissertation will examine the design and implementation of Tornado, and demonstrate how it achieves locality and concurrency in the services it provides. The design decisions of Tornado are evaluated with a mostly complete prototype implementation running on a real scalable multiprocessor, namely NUMAchine [Vranesic *et al.*, 1995], and on a complete machine simulator, SimOS [Rosenblum *et al.*, 1997].

The dissertation begins with an examination of the multiprocessor hardware issues that motivate much of this work, and the basic design principles that come out of these issues. This is followed, in Chapter 3, with a description of the hardware and experimental framework within which the research is conducted, plus the basic operating system structure that underlies Tornado and shapes its design.

The next three chapters discuss a number of the key components of Tornado. Each chapter begins with a motivational section that provides some background material, followed by a description of the design and implementation of the component. Next, the performance of the component is evaluated, and the chapter concludes with a discussion of open issues, related work, and a summary of the chapter as a whole. The first component discussed, in Chapter 4, is the clustered object system, which forms one of the cornerstones of Tornado. Chapter 5 then considers the other key component of Tornado, the protected procedure call interprocess communication facility. In addition to these two components, a number of other support facilities are required for a high performance system, and Chapter 6 examines two of the most important ones: locking and dynamic memory allocation.

Although each chapter includes performance evaluation of the individual components, it is left to Chapter 7 to evaluate overall system performance on a number of benchmarks and compare Tornado to other commercial multiprocessor systems. Finally, we consider the lessons that have been learned from the experience with both Hurricane and Tornado in Chapter 9, and then end with some concluding remarks in Chapter 10.

Chapter 2

Background

The Tornado clustered object model (and the overall Tornado design approach) is heavily motivated by the multiprocessor hardware platform it is targeted for. To help understand the issues the operating system faces from the hardware, this chapter examines the impact of the hardware on system software construction, and presents some of the design principles that we have derived from this hardware/software interaction.

2.1 Multiprocessor Hardware Issues

In this section we describe how shared-memory multiprocessor hardware affects system software. In particular, we consider the effects of (i) true parallelism resulting from multiple processors, (ii) cache coherence and its protocols, and (iii) the high cache-miss latency of shared memory multiprocessors. The characteristics of the small-scale and large-scale multiprocessors we use for this analysis are described in Figure 2.1. There are two main classes of multiprocessors we consider: small-scale bus-based (or cross-bar based) multiprocessors, which commonly range in size from 4-16 processors; and larger CC-NUMA systems (cache-coherent non-uniform memory access), typically built from nodes containing a small number of processors, memory, and disks, and connected by some scalable interconnect (represented here by a cloud), which could be a mesh, tree, torus, etc.

The primary differences between the two architectures are that CC-NUMA systems are able to support larger numbers of processors, memory, and disks, that CC-NUMA systems have variable memory access latencies while smaller systems have fixed latencies, and that CC-NUMA systems are more sensitive to the memory access pattern of workloads which can degrade performance due to contention in the memory subsystem and interconnection network. Despite these differences, however, both architectures have one key thing in common: the performance of both systems is

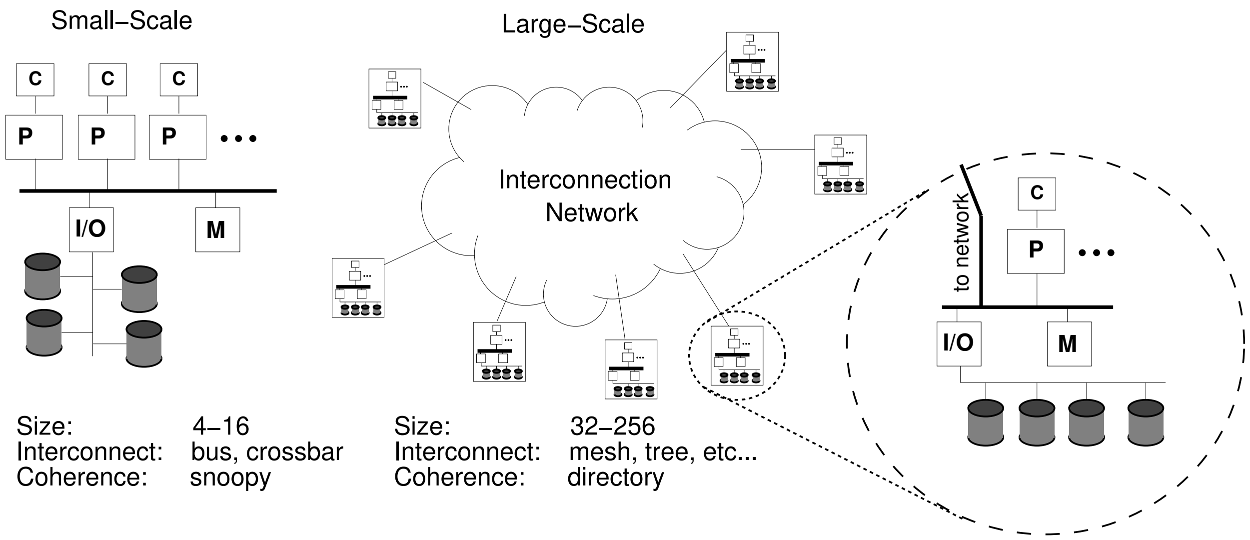


Figure 2.1: *This figure depicts the two basic classes of multiprocessors under consideration. The left figure shows a small-scale bus-based multiprocessor, which commonly ranges in size from 4-16 processors. The figure on the right shows the general architecture of a large-scale system, typically built from nodes containing a small number of processors, memory, and disks, and connected by some scalable interconnect (represented in the figure by a cloud), which could be a mesh, tree, torus, etc. Important features common to both types of systems are the use of an invalidation-based coherence scheme (rather than the less common update-based scheme), and in-cache synchronization primitives (rather than network- or memory-based primitives).*

heavily dominated by caching effects.

2.1.1 Physical parallelism

The most obvious difference between shared-memory multiprocessors (SMMPs) and uniprocessors is the number of processors. Although uniprocessor system software may already deal with concurrency issues, the true parallelism in multiprocessors introduces additional complications that can affect both the correctness and performance of uniprocessor synchronization strategies.

The strategies often employed for synchronization on uniprocessors, such as disabling interrupts in the kernel [Ritchie and Thompson, 1974] or relying on the non-preemptability of a server thread [Hutchinson and Peterson, 1991], are not directly applicable on multiprocessors. Although these strategies can be made to work by allowing only a single processor to be in the kernel at a time (or a single process to be executing in a server at a time), this serializes all requests and is not acceptable for performance reasons. For example, **gcc** spends over 20 percent of its time in the kernel under Ultrix [Chen and Bershad, 1993], which would limit the useful size of a multiprocessor to 5 processors if all kernel requests were serialized. As a result, a fully preemptable (and fully parallelized) system software base is generally required [Eykholt *et al.*, 1992, Talbot, 1995].

Fine-grained locks are generally needed to achieve a high level of concurrency and improved performance. However, the finer the granularity of locks, the larger the number of locks that must be acquired to complete an operation, resulting in higher overhead even if there is no contention for the locks. For example, in a previous system, we found that locking overhead in the fully uncontended case accounted for as much as 25 percent of the total page-fault handling time [Unrau *et al.*, 1994]. It is therefore necessary to carefully balance the desire for high concurrency through finer-grained locks, against the desire for lower overhead through coarser-grained locks.

In addition, there are complex tradeoffs to consider in the implementation of locks on a multiprocessor. Whereas the choice is straightforward for uniprocessors where the only sensible option is a simple blocking lock, multiprocessor locking must consider such issues as context-switch overhead, wasted spin-cycles, bus and network contention due to spinning, fairness, and preemption effects for both lock holders and spinners. Fortunately, this area has been well-studied [Anderson, 1990, Black *et al.*, 1991, Campbell *et al.*, 1991b, Gupta *et al.*, 1991, Karlin *et al.*, 1991, Unrau *et al.*, 1994], and in most cases spin-then-block locks with exponential back-off for the spinning phase have proven to be highly effective [Gupta *et al.*, 1991].[1]

2.1.2 Cache coherence

Software on a shared-memory multiprocessor suffers not only from cold, conflict, and capacity cache misses as on a uniprocessor, but also from *coherence* misses. Coherence misses are caused by read/write sharing and the cache-coherence protocol,[2] and are often the dominant source of cache misses. For example, in a study of IRIX on a 4-processor system, Torrellas found that misses due to sharing dominated all other types of misses, accounting for up to 50 percent of all data cache misses [Torrellas *et al.*, 1992]. In addition to misses caused by direct sharing of data, writes by two processors to distinct variables that reside on the same cache line will cause the cache line to ping-pong between the two processors. This problem is known as *false sharing* and can contribute significantly to the cache miss rate. Chapin et al., in investigating the performance of IRIX ported to a 32 processor experimental system, found that many of the worst-case hot spots were caused by false sharing [Chapin *et al.*, 1995a]. Although strategies for dealing with misses in uniprocessors

[1] Distributed queue-based spin locks are another well-studied option, whose prime value is for the less common cases that can benefit from continual spinning (even under high contention) and which also require low-latency and a high degree of fairness [Magnussen *et al.*, 1994, Mellor-Crummey and Scott, 1991].

[2] To be more precise, invalidation-based cache coherence protocols require that the cache obtain exclusive access to a line the first time it is written to, even if no sharing is taking place. These misses are sometimes referred to as *upgrade* misses or *initial-write* misses. Hence there is an extra cost in multiprocessors both for sharing misses and upgrade misses.

by maximizing temporal and spatial locality and by reducing conflicts are well known, if not always easily applied, techniques for reducing true and false sharing misses in multiprocessors are less well understood. Semi-automatic methods for reducing false sharing, such as structure padding and data regrouping, have proven somewhat effective, but only for parallel scientific applications [Jeremiassen and Eggers, 1995, Torrellas *et al.*, 1994].

The effects of synchronization variables can have a major impact on cache performance, since they can have a high degree of read/write sharing and often induce large amounts of false sharing. For example, Rosenblum noted in a study of an 8 processor system that 18 percent of all coherence misses were caused by the false sharing of a single cache line containing a highly shared lock [Rosenblum *et al.*, 1995]. Even without false sharing, the high cost of cache misses can increase the cost of a lock by an order of magnitude over the fully cached case.[3]

2.1.3 Cache miss latency

Besides the greater frequency of cache misses due to sharing, SMMP programmers are also faced with the problem that cache misses cost more in multiprocessors regardless of their cause. The latency increase of cache misses on multiprocessors stems from a number of sources. First, more complicated controllers and bus protocols are required to support multiple processors and cache coherence, which can increase the miss latency by a factor of two or more.[4] Second, the probability of contention at the memory and in the interconnection network (bus, mesh, or other) is higher in a multiprocessor. Extreme examples can be found in early work on Mach on the RP3 where it took 2.5 hours to boot the system due to memory contention [Chang and Rosenburg, 1992], and in the work porting Solaris to the Cray SuperServer where misses in the idle process slowed non-idle processors by 33 percent [McCrocklin, 1995]. Third, the directory-based coherence schemes of larger systems must often send multiple messages to retrieve up-to-date copies of data or invalidate multiple sharers of a cache-line, further increasing both latency and network load. This effect is clearly illustrated in the operating system investigation by Chapin et al., which found that local data miss costs were twice the local instruction cache miss costs, due to the need to send multiple remote messages [Chapin *et al.*, 1995a].

In the case of large systems, the physical distribution of memory has a considerable effect on performance. Such systems are generally referred to as NUMA systems, or Non-Uniform Memory

[3] Some results suggest that there is often sufficient locality that the locks can generally be assumed to reside in the cache [Torrellas *et al.*, 1992]. However, later work (by the same author) suggests that coherence traffic due to locking is a significant problem [Xia and Torrellas, 1996]. The effects of false sharing may well explain this apparent contradiction.

[4] Even the Digital 8400 multiprocessor, which is expressly optimized for read-miss latency, has a 50 percent higher latency than Digital's earlier, lower performance, uniprocessors [Fenwick *et al.*, 1995].

Access time systems, since the time to access memory varies with the distance between the processor and the memory module. In these systems, physical locality plays an important role in addition to temporal and spatial locality. An example can be seen in the work of Unrau et al., where, due to the lack of physical locality in the data structures used, the uncontended cost of a page fault increased by 25 percent when the system was scaled from 1 to 16 processors [Unrau *et al.*, 1995].

As a result of the high cost of cache misses, newer systems are being designed to support non-blocking caches and write buffers in order to hide load and store latencies. Many of these systems allow loads and stores to proceed while previous loads and stores are still outstanding. This results in a system in which the order of loads and stores may not appear the same to each processor; these systems are sometimes referred to as being weakly consistent [Gharachorloo *et al.*, 1990]. The implication of weakly consistent systems is that regular memory accesses cannot be used for synchronization purposes. For example, if one processor fills a buffer and marks it filled, other processors might see the buffer first being marked filled and then actually filled with data. This could cause some processors to access stale data when they see the buffer marked filled. As a result of this behaviour, software must generally switch from using simple flags to using full locking, with significantly more expensive special memory synchronization instructions. This difference is illustrated by the cost of a lock/unlock pair in AIX on the PowerPC, which is 100 times more expensive than a cached store [Talbot, 1995].[5]

Another result of the high cache-miss latency is the movement towards larger cache lines of 128 or even 256 bytes in length. These large cache lines are an attempt to substitute high bandwidth (which is relatively easy to design into a system) for low latency (which is much harder to design in). Unfortunately, large cache lines tend to degrade performance in SMMPs due to increased false sharing.[6] The example cited earlier of a single falsely-shared cache line being responsible for 18 percent of all coherence misses was due in part to the fact that the hardware had 128 byte cache lines, causing a critical lock to be shared with 22 other randomly placed variables [Rosenblum *et al.*, 1995].

2.1.4 Summary

The memory access times in a multiprocessor span several orders of magnitude due to architectural and contention effects (see Table 2.1). To mitigate these effects, system software must be struc-

[5] In principle, locks would not be required for single flag variables if it were possible to insert (or have the compiler insert) the required memory barrier instructions at the right places, but such support is not generally available at this time [Adve and Gharachorloo, 1995].

[6] To further complicate matters, with a wide variety of cache line sizes (sometimes even within the same product line), ranging from 16 to 256 bytes, it is becoming increasingly difficult to optimize for a fixed or "average" cache line size.

Access Type	Latency
Primary cache	1
Secondary cache	10
Local memory	100
Remote memory	200–500
Hot spot	1000–10000

Table 2.1: *Typical latencies for different types of accesses in a typical large-scale shared-memory multiprocessor, measured in processor cycles.*

tured to reduce cache misses and increase physical locality. In the next section we present a set of structuring techniques and the circumstances under which they should be applied.

2.2 Design principles

The issues described in the previous section clearly have a major impact on the performance of multiprocessor system software. In order to achieve good performance, system software must be designed to take the hardware characteristics into account. There is no magic solution for dealing with the problem; the design of each system data structure must take into account the expected access pattern, degree of sharing, and synchronization requirements. Nevertheless, we have found a number of principles and design strategies that have repeatedly been useful, particularly when applied to clustered object construction (discussed in more detail later in this dissertation). These include principles previously proposed by us and others [Chapin *et al.*, 1995a, Unrau *et al.*, 1994, Unrau *et al.*, 1995, Xia and Torrellas, 1996], refinements of previously proposed principles to address the specific needs of system software [Hill and Larus, 1990], and a number of new principles.

We have partitioned the design principles into so-called $\boldsymbol{S}_*$, $\boldsymbol{L}_*$ and $\boldsymbol{Pl}_*$ principles, described in the following three subsections. $\boldsymbol{S}$ refers to "structuring," $\boldsymbol{L}$ refers to "locking," and $\boldsymbol{Pl}$ refers to "physical locality."

2.2.1 Structuring data for caches

When frequently accessed data is shared, it is important to consider how the data is mapped to hardware cache lines and how the hardware keeps the cached copies of the data consistent. Principles for structuring data for caches include:

$\boldsymbol{S}_{segr\_rmd}$: *Segregate read-mostly data from frequently modified data.* Read-mostly data should not reside in the same cache line as frequently modified data in order to avoid false sharing. Segregation can often be achieved by properly padding, aligning, or regrouping the data. Consider a linked list whose structure is static but whose elements are frequently modified. To avoid having the modifications of the elements affect the performance of list traversals, the search keys and link pointers should be segregated from the other data of the list elements.[7]

S_{segr_rwd}: *Segregate independently accessed read/write data from each other.* This principle prevents false sharing of read/write data, by ensuring that data that is accessed independently by multiple processors ends up in different cache lines.

S_{priv_wmd}: *Privatize write-mostly data.* Where practical, generate a private copy of the data for each processor so that modifications are always made to the private copy and global state is determined by combining the state of all copies. This principle avoids coherence overhead in the common case, since processors update only their private copy of the data. For example, it is often necessary to maintain a reference count on an object to ensure that the object is not deleted while it is being accessed. Such a reference count can be decomposed into multiple reference counts, each updated by a different processor and, applying S_{segr_rwd}, forced into separate cache lines.

$S_{per-proc}$: *Use strictly per-processor data wherever possible.* If data is accessed mostly by a single processor, it is often a good idea to restrict access to the data to only that processor, forcing other processors to pay the extra cost of inter-processor messaging on their infrequent accesses. In addition to the caching benefits (as in S_{priv_wmd}), strictly per-processor structures allow the use of uniprocessor solutions to synchronize access to the data. For example, for low-level structures, disabling interrupts is sufficient to ensure atomic access. Alternatively, since data is only accessed by a single processor, the software can rely on the ordering of writes for synchronization, something not otherwise possible on weakly consistent multiprocessors (see Section 2.1.3).

2.2.2 Locking data

In modern processors, acquiring a lock involves modifying the cache line containing the lock variable. Hence, in structuring data for good cache performance, it is important to consider how accesses to the lock interact with accesses to the data being locked.

L_{priv_rwl}: *Use per-processor reader/writer locks for read-mostly data.* A lock for read-mostly data should be implemented using a separate lock for each processor. To obtain a read-lock,

[7] This is in marked contrast to uniprocessors, where it is better to co-locate the linked list state and the list elements so that when an element is reached some of its data will already have been loaded into the cache.

a processor need only acquire its own lock, while to obtain a write-lock it must acquire all locks. This strategy allows the processor to acquire a read-lock and access the shared data with no coherence overhead in the common case of read accesses. (This principle can be viewed as a special case of principle $\boldsymbol{S}_{priv\_wmd}$.)

$\boldsymbol{L}_{segr\_contL}$: *Segregate contended locks from their associated data if the data is frequently modified.* If there is a high probability of multiple processes attempting to modify data at the same time, then it is important to segregate the lock from the data so that the processors trying to access the lock will not interfere with the processor that has acquired the lock.[8]

$\boldsymbol{L}_{colloc\_uncontL}$: *Collocate uncontended locks with their data if the data is frequently modified.* When a lock is brought into the cache for locking, some of its associated data is then brought along with it, and hence subsequent cache misses are avoided.

2.2.3 Localizing data accesses

For large-scale systems, the system programmer must be concerned with physical locality in order to reduce the latency of cache misses, to decrease the amount of interconnection network traffic, and to balance the load on the different memory modules in the system. Physical locality can be especially important for operating systems since they typically exhibit poor cache hit rates [Chen and Bershad, 1993].

$\boldsymbol{Pl}_{repl\_rmd}$: *Replicate read-mostly data.* Read-mostly data should be replicated to multiple memory modules so that processors' requests can be handled by nearby replicas. Typically, replication should occur on demand so that the overhead of replicating data is only incurred when necessary.

$\boldsymbol{Pl}_{migr\_rw-priv-d}$: *Partition and migrate read/write data.* Data should be partitioned into constituent components according to how the data will be accessed, allowing the components to be stored in different memory modules. Each component should be migrated on use if it is primarily accessed by one processor at a time. Alternatively, if most of the requests to the data are from a particular client, then the data should be migrated with that client.

$\boldsymbol{Pl}_{priv\_wmd}$: *Privatize write-mostly data.* Privatizing write-mostly data, as described in principle $\boldsymbol{S}_{priv\_wmd}$, can be used not only to avoid coherence overhead, but also to distribute data across the system to localize memory accesses.

[8] Special hardware support for locks has been proposed that results in no cache-coherence traffic on an unsuccessful attempt to acquire a lock, making principle $\boldsymbol{L}_{segr\_contL}$ unnecessary [Kagi *et al.*, 1995].

Although the following principles are not strictly for structuring data for locality, we have found them equally important in achieving efficient localization of operating system data.

$\boldsymbol{Pl}_{appr\_local}$**:** *Use approximate local information rather than exact global information.* For certain operating system policies, it is possible to sacrifice some degree of accuracy in exchange for performance by using local approximate information to make reasonable decisions rather than exact global information. For example, having per-processor run-queues reduces short-term fairness globally, but minimizes the cost to the dispatcher when a process is being scheduled to run.

$\boldsymbol{Pl}_{no\_barriers}$**:** *Avoid barrier-based synchronization for global state changes.* When data is replicated or partitioned, it is necessary to synchronize when making global changes to the replicas and when determining a globally consistent value for the partitioned data. In this case, the system should avoid using barrier-based synchronization because it wastes processor cycles while waiting for other processors to reach the barrier, and results in a high overhead in interrupting (and restarting) the tasks running on the processors [Xia and Torrellas, 1996]. There are a variety of alternative *asynchronous* schemes (e.g., as used for lazy TLB shootdown [Peacock *et al.*, 1992]) which can, for example, allow multiple requests to be combined together to reduce the amount of inter-processor communication.

In applying the $\boldsymbol{S}_*$, $\boldsymbol{L}_*$, and $\boldsymbol{Pl}_*$ principles, the programmer must be aware that their over-zealous application may actually reduce performance. For example, while a naive application of the $\boldsymbol{S}_*$ principles may result in a system with reduced coherence overhead, it may also result in an increased total number of cache misses due to the fragmentation of data structures. Hence, significant expertise is required to apply the design principles in a balanced fashion.

2.3 Summary

The main issues facing system software on multiprocessors is the high cost of cache misses and the increased potential for high miss rates due to sharing. Thus the key to improved performance is to reduce unnecessary sharing to a minimum through restructuring that divides and segregates data structures based on the expected access pattern.

The issues and principles raised in this chapter are closely related to the design and implementation techniques discussed throughout this dissertation. In particular, the clustered object approach to system design is based heavily on the principles that relate to localizing data structures through distribution, migration, and replication (principles $\boldsymbol{S}_{per-proc}$, $\boldsymbol{L}_{priv\_rwl}$, $\boldsymbol{Pl}_{repl\_rmd}$, $\boldsymbol{Pl}_{migr\_rw-priv-d}$, and $\boldsymbol{Pl}_{priv\_wmd}$). These principles are used to help determine how to structure a clustered object in terms of representatives (i.e., what data and functionality to partition in what way among the representatives, or whether to have more than a single representative at all). Principle $\boldsymbol{Pl}_{appr\_local}$ (use approximate local knowledge rather than exact global knowledge) appears in many places where local hints are used (as will be seen, for example, in the section on remote protected procedure calls in section 5). Principle $\boldsymbol{Pl}_{no\_barriers}$ (avoid barrier-based synchronization) is also a key component of the design of Tornado, in that we try to make protocols as asynchronous as possible (as will be seen to be important in cluster object destruction discussed later in Section 4). The other principles are used primarily to decide how to structure individual representatives as well as the components that provide the infrastructure that support clustered objects and the rest of Tornado. Hence all the principles form a critical substrate for the design and implementation of Tornado.

Chapter 3

System Overview

This chapter provides an overview of the operating environment and an introduction to the basic structure of Tornado in order to provide the context for a more detailed examination of a subset of the components in subsequent chapters.

3.1 Operating environment

As we saw in the introduction, Tornado is a research operating system whose primary purpose is to investigate operating system structuring issues in large-scale shared-memory multiprocessors, as well as to investigate issues of scalability, flexibility, and application resource control. Other issues of interest in the Tornado research are multi-resource scheduling issues (dealing with memory and I/O demands of applications, in addition to CPU demands), support for memory and I/O intensive large-scale parallel applications, and guaranteed resource delivery to facilitate application optimizations and performance debugging.

Tornado is part of a larger project that involves the design and construction of a novel multiprocessor (NUMAchine [Vranesic *et al.*, 1995], discussed below), an operating system (Tornado), a parallelizing compiler (Jasmine [Abdelrahman *et al.*, 1998]), and various parallel applications. As such, Tornado must be reasonably stable and functional so as to permit production use, while still supporting implementation and policy experimentation.

Tornado is targeted at the general class of large-scale multiprocessors of the form discussed in Chapter 2 (see Figure 2.1).[1] However, because this research is experimentally based, the Tornado implementation is more specifically targeted at the NUMAchine multiprocessor (see Figure 3.1).

[1] Actually, many of the ideas presented in this dissertation would be effective even for small-scale bus-based systems, but to help maintain focus, most of the dissertation will concentrate on large-scale systems.

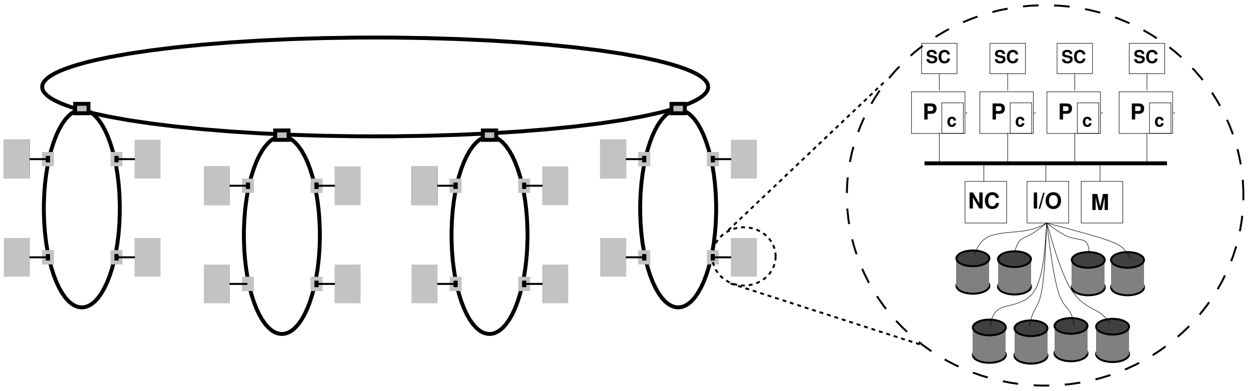


Figure 3.1: *This figure shows the general architecture of the NUMAchine multiprocessor. The system is composed of stations (shown expanded on the right) consisting of multiple processors with two-levels of cache, local disks and memory, and a network cache for remote memory, all connected by a bus. Multiple stations are connected by a hierarchy of high-speed, bit-parallel rings. The architecture is intended to scale up to several hundred processors.*

The NUMAchine architecture consists of a collection of small-scale bus-based *stations* connected by a hierarchy of high-speed bit-parallel rings. The final prototype will have 2 levels of rings (4 local rings connected by a single global ring) connecting 16 stations with four processors, 4-8 disks, and one 128MB memory module per station. From a performance perspective, the remote-memory access time is about twice the local-memory access time, and, with contention in the interconnect, can increase several times further. Hence, memory modules cannot be treated uniformly in terms of access latency.

From a workload perspective, Tornado is intended to support a wide variety of application mixes with a diverse set of operating system requirements. For example, we expect the Tornado workload to include:

- Intelligent parallelizing compiler supported runtime systems: these systems could take advantage of complex and detailed information and control interfaces from the operating system to manage the degree of parallelism of the application, its memory usage, locality, cache reuse, etc.

- Explicitly parallel applications: these applications need high-level interfaces that are easy and intuitive for programmers to use to describe their application resource requirements, such as those described above (parallelism, locality, etc).

- Sequential applications: here, the issue is ensuring that sequential applications perform as well on a large multiprocessor as on a uniprocessor, and that high throughput can be maintained for a large number of such applications. This requires that the resources dedicated to each application be allocated close to the application and managed as independently as possible from the other applications' resources.

- Sequential interactive applications: as above, but here interactive responsiveness is critical as well.
- Parallel application debugging, visualization, and interactive program steering: these types of applications require a combination of the resource management approaches discussed earlier in addition to interactive responsiveness. This complicates resource management since it is not possible to take advantage of the batch mode typically used for large-scale (non-interactive) applications.
- Databases: here the key is to provide the right interfaces that allow database systems to maintain the control they require over resources without interfering with other applications running on the same system; i.e., allowing databases to be good citizens within a shared system.

The purpose of these examples is to demonstrate the wide variety of requirements that the workloads on large parallel systems can have and the demands they are likely to place on the operating system. This in turn, we maintain, argues for a powerful and flexible operating system interface that allows applications to more directly control their physical resources than is possible with existing systems.

3.2 Experimental Setup

The Tornado research is being conducted on both a prototype multiprocessor and a complete system simulator. We describe each platform briefly in turn.

3.2.1 NUMAchine

As mentioned in the previous section, the NUMAchine architecture consists of multiple bus-based stations with up to four processors connected by a bit-parallel ring, with multiple rings connected by a global ring.

For the experimental work in this dissertation, we use a 4 station/16 processor NUMAchine system (the largest currently available). The processors are 150 MHz MIPS R4400 processors (hence with a 6.67ns cycle time) with 16KB direct-mapped instruction and data caches, and a 1MB unified direct-mapped secondary (off-chip) cache. Each station has four processor cards and a memory card with 128MB of memory. The station bus is 64 bits wide and is clocked at 40 MHz, supporting a maximum bus bandwidth of 320MB/s. The stations are connected by a 64-bit wide ring that also

Access Type	Latency
Primary cache	3
Secondary cache	15
Local memory	270
Remote memory	850

Table 3.1: *This table presents the measured latencies for memory accesses to different levels of the memory hierarchy. The latencies are presented in processor cycles (6.67ns per cycle).*

runs at 40MHz and hence also supports 320MB/s of bandwidth. Multiple lower-level rings can be connected by a global ring that is also 64 bits wide, but clocked at 82.5MHz, however the global ring is not currently available. Additional NUMAchine features include support for broadcasting data and interrupts, and a network cache which caches data from remote stations in local station memory (with an access cost similar to that of local memory).

The measured performance of the key hardware components is given in Table 3.1. Of particular note is the relatively high latencies for the secondary cache and main memory. This is due to the use of FPGAs in the prototype. As a result, obtaining good performance is even more challenging on NUMAchine than on comparable commercial multiprocessors. However, although slow in absolute terms, the ratio of processor speed to memory speed for the various components is likely quite close to what will be seen in next generation systems with order-of-magnitude faster processors.

3.2.2 SimOS simulator

An additional important target for Tornado is the SimOS simulator [Rosenblum *et al.*, 1997], which allows us to simulate a NUMA multiprocessor similar but not identical to the NUMAchine architecture.[2] Using a simulator for this allows us to experiment with different sizes and configurations and also collect more detailed statistics about the relationship between the operating system structure and the hardware.

For the base case we configure the simulator to mimic the basic structure of NUMAchine, with four processors per station and cache and memory latencies similar to those of NUMAchine. The resulting system is a reasonable match to that of NUMAchine. Figure 3.2 shows the results of a number of different Tornado micro-benchmarks run on SimOS and on NUMAchine. The tests are discussed in more detail throughout the dissertation. As with most of the graphs in this dissertation, the number of processors is varied along the x-axis while the y-axis shows the average running time (in this case, in microseconds, but sometimes in cycles) for the test across all processes. In general,

[2]In particular, the interconnect and the coherence protocol implementation are different, and SimOS does not support a network cache.

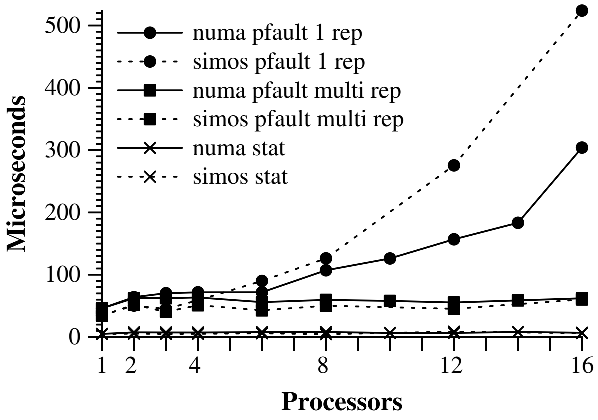


Figure 3.2: *Comparison of SimOS vs. NUMAchine for various benchmarks. "pfault 1 rep" is a concurrent page fault test using a non-scalable underlying implementation of one of the components. "pfault multi rep" is the same test, but with a more scalable implementation. "stat" is a concurrent file stat test. More details on these benchmarks are presented throughout this dissertation.*

SimOS tends to underestimate the time for short, cache-friendly codes, and over-estimate the time for highly contended, memory-system intensive codes. As a result, the measured times between SimOS and NUMAchine are not identical, but the general trends match relatively closely, allowing us to reasonably evaluate the effect of various tradeoffs in the Tornado architecture using SimOS.

3.3 Tornado Architecture

In this section we look at the architecture of Tornado in more detail. It is important to note, however, that most of the design details that follow here and throughout the dissertation are concerned with how Tornado *should* be implemented, not necessarily its current state. As usual with research prototypes, time constraints prevented some features from being fully implemented. Where a design feature is described that has not been fully implemented, the degree of completeness of the implementation and the expected complications of an actual implementation will be discussed.

3.3.1 Operating System structure

Tornado is a fully functional, multi-user operating system, providing a Posix-like environment that allows many Unix utilities to be ported with a simple recompilation. Although it provides a Posix-like interface for compatibility, its native interface and structure is quite different from that of most Unix operating systems.

First, Tornado is a microkernel system, meaning that only key features are provided by the kernel, with the rest provided by user-level servers. The Tornado kernel provides memory manage-

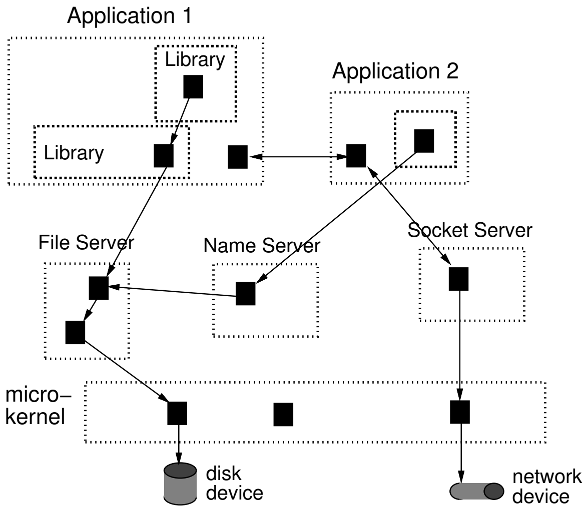


Figure 3.3: *The Tornado operating system is shown here broken down into some of its constituent parts: the microkernel, servers, libraries, and applications. The kernel, servers, and applications all reside in separate address spaces. The black squares represent individual objects in the different address spaces, with the lines connecting them representing paths of communication.*

ment, process management, interprocess communication, and exception handling. Most other features, such as terminal support, pipes, networking, local and networked file systems, and debugging support, are provided by user-level servers and/or libraries linked into client applications (see Figure 3.3).

Second, Tornado is built on an object-oriented substrate: both the internal construction of the kernel and servers, and the way they export resources to clients, make use of object-oriented facilities and models. Objects in the system range from low-level components, such as process descriptors, page-caches, and address translation objects, to more directly visible objects such as files, directories, sockets, and pipes. Clients communicate with servers by invoking operations (or methods in C++ parlance) on these objects. For example, opening a file involves a call to a directory object in which the file resides, which returns a reference to an open-file object supporting such operations as read, write, and close.

Finally, interprocess communication (the facility that allows clients to request services from objects in different servers), is based on the Protected-Procedure Call (PPC) model of communication. With PPC, rather than sending messages to waiting processes as in many other microkernel systems, a client makes a call into another address space as if the client's process actually crossed into the server's domain and invoked the corresponding method on the target object directly.[3] This

[3] As we shall see, the actual mechanism involves creating a new thread in the server to handle the request, and terminating the thread on returning from the call.

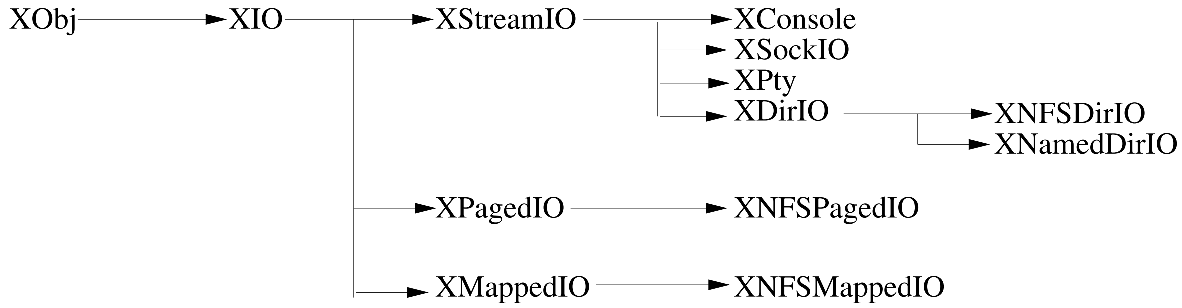


Figure 3.4: *The External object (xobject) class hierarchy for I/O classes is displayed in this figure. Classes to the right inherit from those on the left. Classes positioned vertically with respect to one another are sibling classes.*

provides a number of benefits, particularly on multiprocessors, as will be detailed later in chapter 5.

3.3.2 Tornado programmer interface

Access to the Tornado kernel and servers is generally provided through objects.[4] Objects that allow external access (to exported system services) are called *external objects*, or *xobjects*. Xobjects are organized as a public class hierarchy to facilitate the use of standard interfaces.[5] An example of the use of the xobject hierarchy to define generic and specific interfaces to system services is show in Figure 3.4.

In the figure, a subset of the hierarchy related to I/O is shown. Classes to the right are inherited from those on the left. Classes positioned vertically with respect to one another are sibling classes. Classes for entities such as network connections and ttys are subclasses of the XStreamIO class, since they provide a streaming interface. Since files are generally accessed using memory mapped I/O, two interfaces are provided: one, XMapppedIO, defines the generic interface for objects that can be mapped into a program's address space; a second interface, XPagedIO, is used by the kernel to move pages of the object into and out of memory as a result of page faults by the application.

Applications typically obtain a reference to an xobject from one of four main sources: (*i*) from their creator through inheritance; (*ii*) from the name server as the result of a name lookup on some object; (*iii*) from the result of a C++ **new** call on an xobject's class;[6] or (*iv*) from some other program through an explicit transfer.

[4]Tornado currently only supports C++ object interfaces.
[5]Servers are of course free to define their own classes or specialize one of the existing classes.
[6]The **new** is actually performed on the xobject's proxy class, to be discussed in Chapter 4.

3.3.3 Basic programmer-visible abstractions

The Tornado kernel exports a relatively small set of abstractions, most in the form of objects.[7] A program in Tornado is the basic unit of resource allocation, and corresponds to what one would normally consider a process in Unix systems. It consists of an address space, a set of virtual processors (described below), a set of threads of execution, which we call processes, that execute within the address space and are associated with one of the program's virtual processors, and a set of ports that are the targets of PPC calls.

Virtual processors represent the unit of processor allocation and locality. It provides an abstraction for the programmer, allowing data and processes to be related through the virtual processor by specifying distribution or locality requirements in terms of the virtual processor. For the system scheduler, it provides a unit for allocation and a hint for the co-scheduling of memory and processors.

A program's address space consists of a set of contiguous page-aligned regions that typically correspond to page-aligned portions of files (including backing store files for anonymous memory). Memory mapped I/O is the prime method of accessing files (although a more traditional I/O interface, provided by the ASF library [Krieger *et al.*, 1994], is available for compatibility).

A program's ports represent entry points that are available as targets of PPC calls. Most programs have just a single port that handles accesses to all objects exported by the program. For servers, these exported objects generally represent the services they provide, such as file objects for a file server, or socket objects for a network server. However, all programs also export special objects for facilities such as signal handling and debugging (with the appropriate security constraints on their use).

Finally, each program is assigned a *badge*, which uniquely identifies it. The badge is primarily used for authentication, allowing a server to quickly identify a client as one that has been previously authorized to access some specific service. (Traditional user-ids are used for the initial authentication.)

More details concerning the PPC facility and ports and badges will be presented in Chapter 5.

3.3.4 Key kernel classes

The Tornado kernel is decomposed into a number of basic classes that provide the fundamental facilities of the kernel. Each component, or building block, has a well defined interface, specified by a C++ abstract class. This allows multiple implementations for each building block of the system,

[7] A few of the abstractions, as we shall see later, are represented by basic types, such as the virtual processor abstraction which is normally referred to simply by an integer identifying the particular virtual processor number.

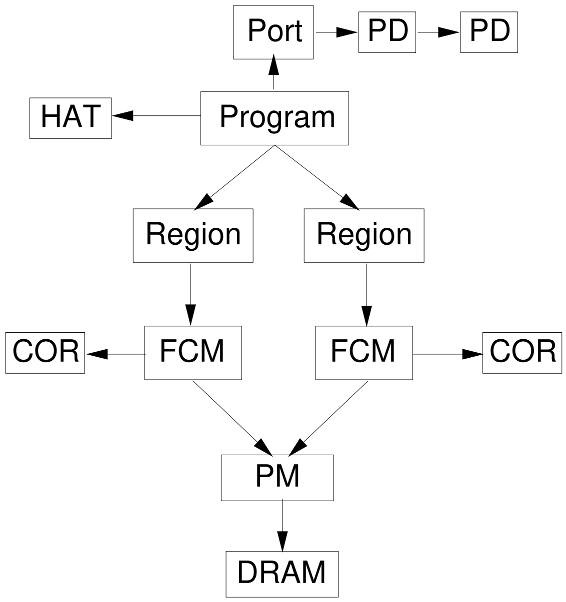


Figure 3.5: *This figure shows the key kernel objects and their functional relationships. In this example a single Program is shown with a Port and two process descriptors (PDs) associated with the Port, a hardware address translation (HAT) object (abstracting the multiple HATs normally associated with a Program), and two Regions. The Regions in turn connect to a file cache manager (FCM) each (they map different objects) which are associated with their cached object representative (COR) and a page manager (PM) responsible for managing the amount of physical memory given to this Program. The PM in turn is connected to the global free physical memory manager (DRAM) that coordinates the various PMs.*

and it allows these building blocks to be composed dynamically, as described later in Section 3.3.5. A brief description of the main object classes follows, with the relationship between the different objects illustrated in Figure 3.5.

Program The Program class is the central hub for all information associated with an individual program, including memory management and process related information, as well as security information such as user and program identifiers. The program object is not directly responsible for most of these resources, but instead is a central directory for finding the appropriate object. Hence its implementation and role are quite simple.

HAT The hardware address translation (HAT) class is responsible for managing the hardware memory management unit (MMU) resources associated with a segment of virtual memory of a given program. It records the virtual-to-physical mappings and interacts with the processor's MMU to establish the mappings in the processor. Although a number of different organizations are possible, Tornado currently uses a two-level scheme, with the lower level consisting of a set of *segment*

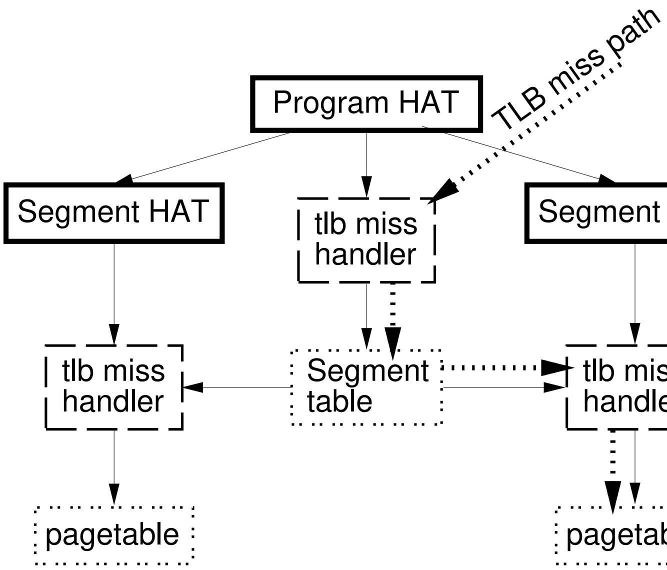


Figure 3.6: *This figure shows the key HAT objects and their functional relationships. The thickly outlined boxes represent the HAT clustered objects, the dashed boxes represent TLB miss handlers, and the dotted boxes represent miss handler mapping tables. The logical path taken by a TLB miss is shown by the thick dotted arrows.*

HAT objects that are responsible for the segments[8] of the virtual address space, and the upper level consisting of a single *program HAT* object that keeps track of all the segment objects of a program (see Figure 3.6). Different segment HAT object subclasses exist for handling different situations such as aliased memory (virtual memory which maps to different physical memory on different virtual processors), multiple page sizes, and special regions optimized for the interprocessor communication facility.

The fact that our hardware supports software TLB miss handling allows us to organize the components of the system that handle TLB misses in a unique, highly customizable fashion. Each program HAT registers with the Program a piece of code for handling TLB misses for the program. Similarly, each segment HAT registers with the program HAT a piece of code for handling TLB misses for its segment of virtual memory. When a TLB miss happens, control is passed to the program HAT's piece of code, which looks up the segment code in a table (based on the address of the TLB miss) and passes control to it; the segment HAT code in turn looks up the pagetable entry for the address and loads the TLB. Because each object supplies its own code, different TLB miss handling routines optimized for different situations are possible, such as large page size support or special-case memory such as the interprocessor communication regions. Although at first glance

[8] The address space is divided into equal-sized chunks, called segments. This division is orthogonal to the division into regions described below.

this might appear prohibitively expensive, by specializing the TLB miss handling code for each HAT object as is needed, replacing variables with constants in the instruction stream, and by using a simplified calling convention, the cost can be reduced substantially. As a result, a typical TLB miss requires approximately 48 instructions, which although somewhat above average for current systems [Jacob and Mudge, 1998], provides far greater flexibility and opportunity for optimizations according to the specific memory access pattern.

Region A Region in Tornado is a contiguous page-aligned portion of a program's virtual address space that is mapped to a contiguous page-aligned portion of a file (or file-like object). Binding Regions to files is the primary means used to access data on secondary storage in Tornado. Regions handle page faults by requesting the appropriate file blocks from the page-caching object and establishing virtual-to-physical mappings for the pages through calls to the HAT. Like HATs, regions come in various flavours. In addition to regular regions, there are regions that support aliased mappings, where the same virtual address in a program may map to different parts of a file depending on the processor which accesses it, and regions that support copy-on-write semantics.

COR The cached object representative (COR) is an in-kernel stand-in for a file-like object provided by a server. There is one COR instance for each object recently mapped into some program's address space. The COR's primary purpose is to serve as an interface between the kernel and the server. Hence there is one class of COR for each type of server that provides mappable objects. Currently there is a CORhfs class for talking to the Tornado file system, a CORnfs class for talking to the Network File System, a CORphys class for mapping physical memory or devices into an address space, and a CORzero for mapping the logical zero device which provides zero-filled pages (e.g., for the BSS section of a program's executable). Regions map objects by talking to the appropriate COR, that in turn provides the region with a cache manager reference (known as an *FCM*, described below) to talk to. The region can suggest a particular FCM to the COR if it wants to use a specific cache management policy, but it is up to the COR to decide whether to honour the request (depending, for example, on whether or not it conflicts with requests by other regions currently mapping the same COR).

FCM The file chunk/cache manager (FCM) is responsible for managing a cache of pages for a COR. It implements local (per FCM) page replacement policies and coordinates among the regions that map the same associated COR. Regions request cached pages from the FCM and the FCM forwards the request on to the COR if it does not have the page cached. The FCM is also responsible for informing the affected regions when a page must be removed (which in turn inform the appropriate HATs to remove the virtual-to-physical mappings associated with the page). There is a one-to-one correspondence between FCMs and CORs, however the FCM associated with a COR can change over time, as the particular cache management policy desired by the COR or the regions mapping it may change. Although the intent is to have many different types of FCMs to support different cache management policies, only two (LRU and FIFO) have currently been implemented.

PM The page managers (PMs), are responsible for managing physical memory for the entire system to ensure a fair distribution of memory across all applications and FCMs. Each program is associated with a PM that controls the amount of memory given to all the FCMs the program is mapping.[9] PMs are organized in a hierarchy, allowing different page allocation and reclamation policies for individual programs and collections of programs (such as all those belonging to a user or a user's session).

Port The Port object is used as the endpoint for interprocess communication. A program can have one or more ports to which other processes send their requests. All processes in a program are created as a side-effect of a call to a port. More details are provided in Chapter 5.

PD Process descriptors (PDs) encapsulate the state of a process (also known as a kernel thread in some systems), including the virtual processor it belongs to, the port it belongs to, and some low-level machine-specific information. A separate class is provided for kernel and user-level processes to support special requirements of kernel processes.[10]

3.3.5 Kernel and system server design strategies

The internal structure of the kernel (as well as the system servers) can be viewed along three main lines: (*i*) kernel decomposition for localized resource management; (*ii*) building blocks for flexibility; and (*iii*) clustered objects for scalability. Each is discussed in turn below.

Kernel decomposition

As discussed earlier, operating systems are driven by the requests of applications on virtual resources such as virtual memory regions, network connections, threads, address spaces, and files. To achieve good performance on a multiprocessor, requests to different virtual resource should be

[9] When more than one program is mapping an FCM simultaneously, the situation is a little more complicated, but such a discussion is beyond the scope of this dissertation. See [Wilk, 1997] for more details.

[10] This is primarily required to avoid certain circular dependencies in the system, but also provides improved performance since kernel-specific processes can take advantage of certain special features that go along with operating in kernel mode.

handled independently, that is, without accessing any shared data structures and without acquiring any shared locks. One natural way to accomplish this is to use an object-oriented strategy, where each resource is represented by a different object in the operating system.

The key kernel data structures presented in the previous section and illustrated in Figure 3.5, present a good example of the advantage of employing an object-oriented approach. In the performance critical case of an in-core page fault, all objects invoked are specific to either the faulting process (the HAT, the Program, and the Region) or the file(s) backing the memory being accessed (the FCM). The locks acquired and data structures accessed are internal to the individual objects and hence independent from other system components. Hence, when different processes are backed by (logically) different files, there is no potential source of contention. Also, if processes run on different processors, the operating system will not incur any communication misses when handling their faults. In contrast, many operating systems maintain a global page-cache, and page faults by different applications contend for the data structures of this page-cache. Moreover, a global page-cache has no locality, and hence even in the absence of contention, the operating system will typically incur the overhead of communication misses when traversing this data structure.

Localizing data structures in the Tornado fashion results in some new implementation and policy tradeoffs. For example, without a global page-cache, it is difficult to implement global policies like a clock replacement algorithm in its purest form [Wilk, 1997].[11] Memory management in Tornado is based on a working set policy (similar to that employed by NT [Custer, 1993]), and most decisions can be made local to FCMs.

In Tornado, most operating system objects have multiple implementations, and the client or system can choose the best implementation at run time. In the future we expect to be able to dynamically change the objects used for a resource. This flexibility is an important tool in solving a number of problems that arise in a multiprocessor environment. For example, it provides us with a mechanism to easily extend the system, to add functionality, adapt the system to new hardware, or improve performance.

One of the greatest benefits of Tornado's object-oriented structure is that it greatly simplifies Tornado's implementation, allowing us to initially implement subsystems using only simple objects with limited concurrency, improving the implementation of the objects only when performance (or publication) requires it. Moreover, the implementation of an object can be specific to the degree of sharing, so implementing an object with locking protocols and data structures that scale is only necessary if the object is widely shared.

[11] On the other hand, many modern operating systems have already abandoned or are in the process of abandoning such policies. For example, with the large amounts of physical memory available on modern systems, AIX and SCO Unix have both moved towards per-file replacement policies to avoid having to traverse large numbers of physical page descriptors that are never paged.

One disadvantage of our approach is the space inefficiencies that result from partitioned management of resources. For example, in a system with a single global page cache, the hash table used to access the cache can be optimized based on the total amount of memory in the system, while it is more difficult to size such data structures when there are many smaller page caches.

Building blocks

System software provides applications with abstractions of virtual resources, such as virtual memory, network connections, files, and processes. In Tornado, each virtual resource instance (e.g., a particular file, open file instance, memory region) is implemented by combining together a set of what we call *building blocks* [Auslander *et al.*, 1997]. Each building block encapsulates a particular abstraction that might (*i*) manage some part of the virtual resource, (*ii*) manage some of the physical resources backing the virtual resource, or (*iii*) manage the flow of control through the building blocks. Building blocks are implemented as clustered objects and thus contain state and export a well-defined interface.

A building-block object exports an interface that specifies the operations that can be invoked by other objects. It may also import (one or more) interfaces that are exported by other building-block objects. Two building blocks are said to be *connected* if one of the building blocks has a reference to the other, and hence can invoke methods on it. Two building blocks may be connected only if the exported interface of the one is imported by the other.

The particular composition of building blocks that implement a virtual resource (i.e., the set of objects and the way they are connected) determines the behaviour and performance of the resource. As a simple example, Figure 3.7(a) shows four building-block objects that implement a file.[12] In the figure, the Compression object compresses and decompresses the stream of data, passing it on to a RAID 0 object that stripes the data over two disks, driven by disk driver objects Driver A and Driver B. The imported and exported interfaces are indicated by the patterned rectangle at the top and bottom of each object. If two building blocks are connected then the corresponding imported and exported interfaces must match.

It is important to note that *each* virtual resource instance will have a different building-block composition. Thus, two open file instances will be implemented by a different set of building blocks, possibly with a different topology, making it possible to offer highly customized services. In our system, it is the application that specifies the composition of resources created on its behalf. Moreover, the composition is dynamic and can, in principle, be changed repeatedly by the application (assuming interface requirements are respected).

[12]The building block approach to system construction was first proposed and developed as part of the Hurricane file system [Krieger, 1994].

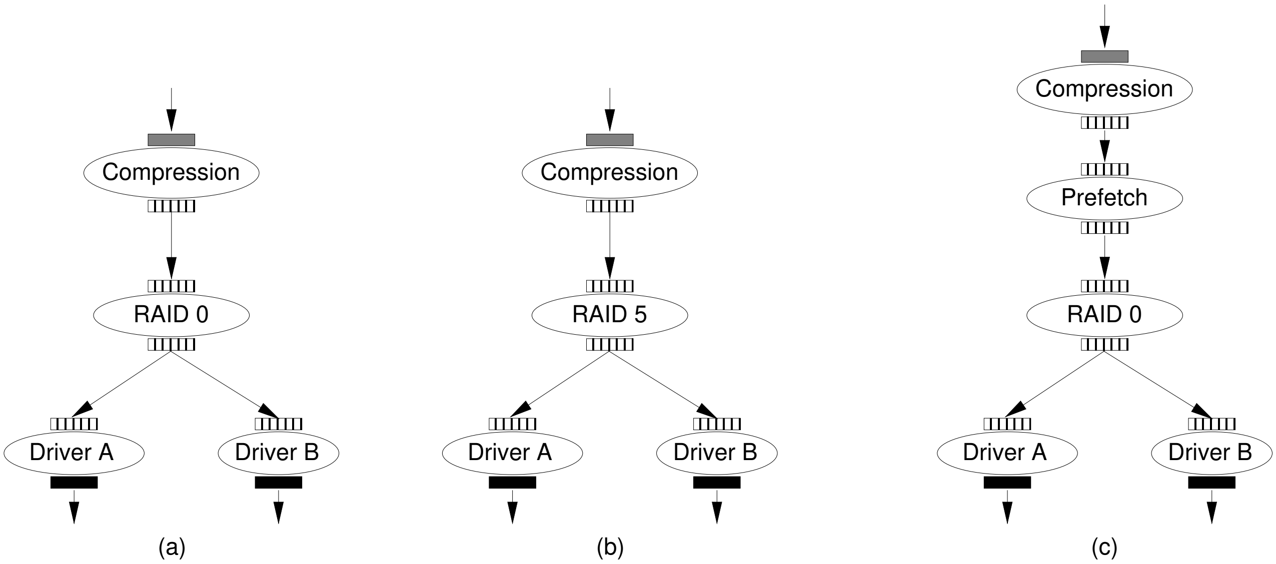


Figure 3.7: *Figure (a) depicts a simple building block composition for a file that uses compression and is striped across two disks. Figure (b) shows how building blocks can be interchanged by, in this example, replacing the RAID 0 striping object with a RAID 5 striping-with-parity object. Finally, Figure (c) shows how the composition can be extended, in this case by adding a prefetching object in between the compression object and the RAID 0 object.*

In our building-block framework, flexibility can be achieved in a number of ways. First, given a particular composition, it is possible to exchange one building block for another as long as the interfaces of the two are the same. For example, in Figure 3.7(a), the RAID 0 striping object can be replaced by a RAID 5 striping-with-parity object, yielding Figure 3.7(b). Thus for each type of building block, multiple implementations may exist, each supporting a different policy or optimized for a different application behaviour. In practice this is achieved by having multiple subclasses provide separate implementations with identical interfaces inherited from a common superclass. Even with only a few subclasses, the combinatorial effect on the behaviour of an entire composition can be huge.

Second, new building blocks can be added to an existing structure if the connecting interfaces match, thus modifying the topology. This can be used to add new functionality. For example, Figure 3.7(c) shows how a Prefetching object can be inserted between the Compression and RAID 0 objects. This is possible because the Prefetching object imports and exports the same interface. These types of building blocks (those that import the same interface they export) can be arbitrarily stacked.

Finally, it is possible to support new interfaces to applications by introducing new building-block objects that export these interfaces, but import existing interfaces so that they can be connected to existing structures.

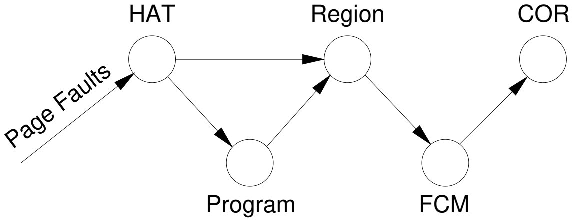


Figure 3.8: *This figure illustrates the basic memory management components that a page fault passes through.*

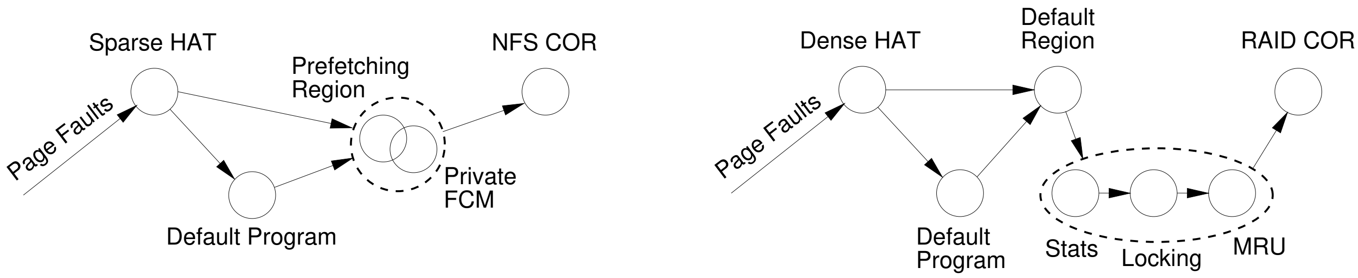


Figure 3.9: *This figure shows two examples of a memory management subsystem built from building blocks. The leftmost figure shows an example of a compound object, where the Region and FCM have been combined into a single object. The rightmost figure shows an example of an expanded object, where the FCM has been formed from three separate FCM sub-components.*

In general, the finer the granularity of the building blocks used in a composition (and thus the larger the number of building blocks in the composition) the larger the degree of flexibility in offered customizability. In the case of the file system, we have found that we tend to use many fine-grained building blocks in a composition, as opposed to using a few large ones. For example, the RAID 0 object above might execute only 2–3 lines of C code in a typical flow of control through the object. Similarly, the larger the number of building-block classes with identical interfaces, the more flexibility in (re-)defining compositions. This is particularly the case when building blocks export the same interface they import.

An important consideration is that, in many cases, it is necessary to provide some structure for the composition of building blocks to make it easier to design new building blocks that are composable with existing ones. In practice, a structure often emerges naturally from the functional requirements of a subsystem, such as the decomposition of the memory management subsystem into the HAT, Program, Region, FCM, and COR classes (see earlier Figure 3.5). Given such a structure, the building block approach is used to obtain flexibility within each component, with standard interfaces used for connecting building blocks across component boundaries.

A simplified version of Figure 3.5 is shown in Figure 3.8 to illustrate the path a page fault takes. The path begins at the HAT, and then passes either first through the Program object or directly through the Region object for the region in which the page fault occurred (depending on the type of HAT and Region objects chosen). The request then passes through the FCM and then (assuming the request cannot be satisfied from the FCM's page cache) through the COR object which in turn passes the request on to the appropriate file server.

Given such a structure, a couple of the options available when applying the building block approach to the memory manager are depicted in Figure 3.9.[13] The leftmost figure shows the case where a HAT object was chosen that optimizes for a sparsely populated address space; a region was chosen that does prefetching and is combined with an FCM optimized for private memory regions to form a compound object (a tightly coupled pair of objects); and a file system interface was chosen to interact with an NFS file server. The rightmost figure depicts a case where a HAT optimized for densely populated address spaces was chosen, coupled with some default region implementation, which is attached to an FCM that has been expanded into three interconnected objects: a statistics collecting object, a locking object to handle atomic transactions, and an MRU page-replacement object. Finally, the far right object interfaces with a RAID file system.

Although the basic principles of building block composition have been fully applied in the Hurricane File System (HFS) (subsequently ported to Tornado), only the first approach to flexibility (choosing specific implementations for a given object) is fully implemented in the Tornado kernel at this time. Applying the second principle, namely integrating several objects or expanding a single object, and integrating it with the clustered object approach is part of ongoing research and beyond the scope of this dissertation. There are several reasons why the application of the building block approach to the Tornado kernel raises new questions not seen in HFS. First, in the file system, I/O overheads dominate the other costs, so building block overheads are insignificant; it remains an open question whether this will be true for the often simple operations performed in the kernel. A second open question is the degree of flexibility that can be achieved in the kernel compared to the file system where there are many simple policies that can be stacked arbitrarily due to the simple read/write I/O interface. A third issue is the potential conflict between composing objects for functionality (the building block approach) and composing objects for internal structuring (the clustered object approach); there may be significant limits to the possible ways the two kinds of composition may be applied. Finally, there is the problem of hidden dependencies not visible in the calling interface—such as locking protocols which may require that certain locks be held or not held at the point of a call—that may be difficult to standardize in such a way that they remain efficient while still supporting arbitrary composition.[14]

[13]Not all of these options are actually implemented, for reasons that will soon become clear.

[14]Clustered objects partially address this issue, as we shall see.

Clustered objects

The third and final aspect of the internal design structure of the kernel and servers is the application of clustered objects. Clustered objects provide the infrastructure for scalable system design and are discussed in greater detail in Chapter 4.

3.4 Summary

In this chapter we described the environment in which this research takes place, as well as the fundamental components of the Tornado kernel and its internal design. In particular, we identified the value of using an object-oriented approach for enhancing locality and concurrency as well as three key additional design strategies: system decomposition into key components, component decomposition into building blocks, and building block decomposition into clustered objects.

Chapter 4

Clustered Objects

In this chapter we examine the core structuring component of Tornado that allows the system to replicate, migrate, and distribute objects in order to handle high contention for individual virtual resources.

4.1 Motivation

As we saw in Chapter 2, a number of issues related to performance arise that are either particular to, or more pronounced in, large-scale multiprocessor systems, such as NUMAchine. First, remote memory accesses have much greater latency than local memory accesses, leading to longer processor stall times. Second, the potential for memory and network contention increases as more processors concurrently access shared data. Third, blind reliance on cache-coherence to solve all locality issues can result in excessive cache-coherence network traffic and congestion at the cache controllers. Finally, false sharing at the cache-line level can compound problems in all parts of the system, even with good cache locality.

A classic example that illustrates some of these problems is the process dispatch queue. Consider the case where a single linked list is used to enqueue and dequeue runnable processes, and whose head and tail pointers are stored in one memory module. The number of updates to this list increases in proportion to the number of processors. If the pointers are left uncached, the memory module will quickly become a bottleneck. If, on the other hand, they are cached, coherency traffic will lead to network congestion and will likely make things worse (see Figure 4.1). Dealing with such locality issues often involves certain tradeoffs, since potential solutions may entail changes to the semantics of the object. In this particular case, if the dispatch queue were partitioned among the processors of the system so that every processor had its own dispatch queue, it might become difficult to respect system-wide priority requirements.

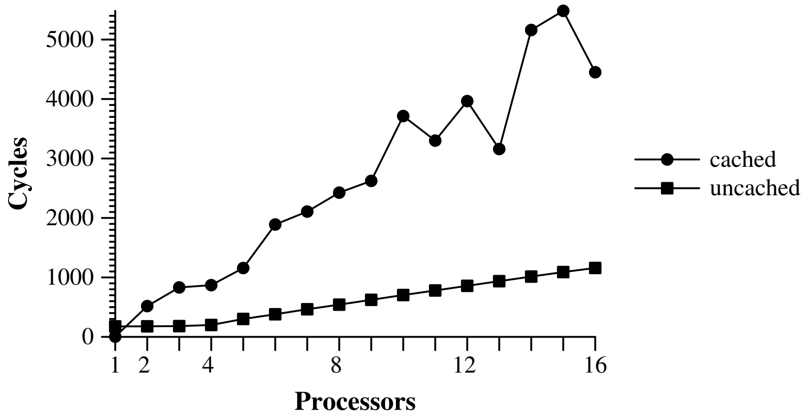


Figure 4.1: *This graph illustrates the effects of contention for a common shared variable. The values measured are cycles per iteration with the loop overhead factored out. In this case, the variable is just read and written; no locks are used. As a result, it is optimistic as to the effects of contention. Note how the cached version is actually slower than the uncached version for all cases except for a single processor (where there is no sharing), due to the cacheline bouncing back and forth between the caches.*

To achieve the best possible performance, it is necessary to distribute system data—through replication, migration, or some form of partitioning—across the system memory in a way that minimizes both contention and the number of remote memory references. Such techniques have been studied in isolation for many problems in operating systems, such as synchronization, memory allocation, and scheduling. For example, contended spin-lock performance is improved by restructuring the lock as a queue of waiters, all spinning on a local flag, thus eliminating the contention effects of a central spinning approach [Mellor-Crummey and Scott, 1991]. Similarly, performance of a multiprocessor memory allocation facility can be significantly improved by replacing the traditional central pool of free memory with per-processor pools that reduce lock and memory contention for the common case, as well as improve cache reuse and reduce cache interference among the competing processors [McKenney and Slingwine, 1993]. One goal of Tornado was to provide a general-purpose framework that would facilitate the application of these techniques within a single coherent framework so as to encourage its use in all layers of the system.

Most of the approaches for dealing with the problems of large-scale NUMA multiprocessors outlined above fall into three broad categories:

- An object can be migrated in response to an expected future access pattern. For instance, a process descriptor might be migrated to the location where the process is running in order to minimize the number of remote memory references during context switches.

- An object can be replicated across the system, relying on software-based update or invalidate protocols to ensure consistency when it is modified. This approach is attractive for read-mostly objects, such as a program object for a parallel program whose address space structure is stable over an extended period of time. In this case the locality benefits obtained by having multiple replicas would likely outweigh the costs of creating and managing the replicas.

- An object can be partitioned and distributed across the system in a way such that each processor most often accesses and modifies data local to it. This approach might be used to distribute the dispatch queue so that processors access remote queues only when there is no local work remaining.

Each object in the system will, in general, require a different combination of techniques. For example, a program object of a parallel program might be replicated across the set of processors the program is scheduled on, but might choose to only partially replicate the region list that describes the virtual address mappings of the program, reflecting the parts of the address space each process of the program has actually accessed. Additionally, although the list of memory regions associated with the address space might be (partially) replicated, the file cache manager (FCM) objects to which the region objects point will likely require a different strategy, since the contents of each individual FCM will change frequently as pages are brought in and ejected with each page fault. The best strategy will depend on the region's access pattern, such as whether it is accessed by a single processor or by a group of processors, and in the latter case, whether each processor accesses a private part of the region or all processors share the region in its entirety.

Although it is important to hide the internal structures and strategies of an object, it is also important for good performance to allow client requests to be efficiently directed to the most appropriate component of a distributed object. Returning to the previous example, although the region list may be replicated, the Region objects themselves may choose a centralized implementation (since they are primarily read-only). However, the file cache manager (FCM), with which the Region communicates, may choose to partition itself in order to keep information about each page close to the processor accessing it (or perhaps the memory holding it). Since, from the Region's point of view, the FCM is a single object, it is not clear to what object the Region should point. One option might be for all references to point to a single front-end component of the FCM which redirects requests to the appropriate component of the FCM, giving the FCM control over the distribution policy. Unfortunately, if implemented in the most straightforward way (see Figure 4.2), this structure would itself become a bottleneck and eliminate many of the benefits expected from partitioning the FCM in the first place. Ideally, we would like the appropriate FCM component to be invoked automatically based on the processor that is referencing the object while always using a common reference for the FCM across all processors.

In summary, our clustered object system must provide three key features:

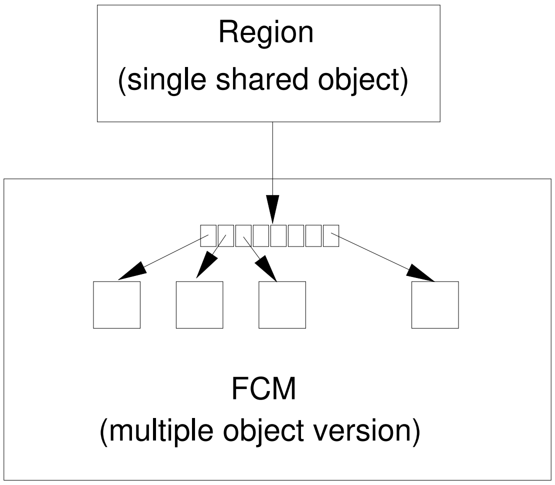


Figure 4.2: *In this figure, we illustrate a potential approach where all references to an FCM refer to a single object that employs a redirection table to forward requests to the component object local to the requesting processor. In this case, the table is a centralized resource that leads to remote memory references on each method invocation, and is therefore a potential bottleneck.*

- support for distributed object structures and policies including migration, replication, and partitioning of a single logical object;
- efficient access to the most appropriate component of a distributed object (where appropriate generally means most local);
- transparent access to this distributed object from a client object through a single reference that can be arbitrarily shared across the system.

The next section describes how the clustered object mechanism achieves these goals by using a distributed redirection table for all objects across all processors.

4.2 Implementation

The clustered object mechanism provides a general framework for objects to hide their internal representation from client objects, yet allow methods to be directed at localized portions of it. Externally, a clustered object is like any other object in that it supports a single, well-defined interface. Internally, the object can migrate, replicate, or distribute data as needed for performance. The way this data is arranged and accessed in the system is what we term the object's *clustering structure*.

Figure 4.3 illustrates the different components of a clustered object. A clustered object consists of multiple *representative objects*, or just *reps*, to which method invocations (i.e., C++ function calls) on the clustered object may be directed. Typically, these representatives are distributed across

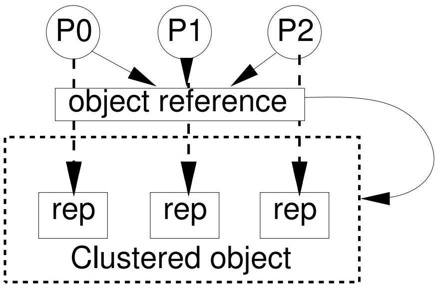


Figure 4.3: *One clustered object consists of all its reps, in this example one per processor over three processors. The Object ID (OID), or object reference as it is often called, is used to refer to the clustered object as a whole, and is converted to a reference to one of the reps when a client uses the OID to invoke a method of the clustered object.*

the system, each providing a (local) point of access for a nearby set of processors (hence called local representatives, or local reps). To reduce the representative creation costs in large systems, clustered objects typically create their local reps on demand as they are needed. Although it is the full collection of reps that together define the implementation of the clustered object, each local rep supports the full object interface, and thus appears to client objects as the complete clustered object. The clustering structure of an object is thus completely hidden from a client, which always refers to the clustered object in a uniform manner using a single object identifier (OID).

As a simple example of a clustered object, consider an object, such as a process descriptor, which contains some long-lived read-mostly data which might be queried on any processor (such as the user id), and some read-write data that is primarily accessed by the processor the process is running on (such as the saved register contents). A possible clustering approach would be to maintain a single copy of the read-write data that is migrated with the process it represents, while replicating the read-mostly data on demand to the other processors accessing the process descriptor (see Figure 4.4). When a client makes a call to the process descriptor, it uses the universal object ID (OID) to refer to the object, which is then translated by the clustered object system to the local rep for the given processor. If the request needs to access the read-write state and the local rep contains only the read-only state, the rep (and not the caller) must locate this state (through shared memory or a form of message passing to be discussed later) in order to complete the operation.

The degree and type of distribution of a clustered object's representatives may vary widely depending on the type of object and how it is accessed, as well the architecture of the multiprocessor. For example, consider the Region object which is used to map a portion of an address space, and is responsible for handling page faults. Typically, its state is frequently accessed but infrequently modified, so it may be sufficient to create only a single rep for some collection of nearby processors (such as the set of processors in the lower-level ring of NUMAchine) since there is little risk of

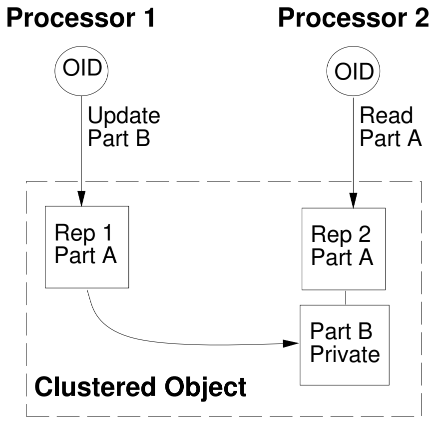


Figure 4.4: *This is an example of a clustered object which has data that is frequently read (Part A), and data which is frequently modified (Part B). The object has two reps, one on each processor, to which local requests are directed. Part A is replicated, so both reps contain copies of Part A, thus localizing access to this data. On the other hand, Part B is not replicated. If an operation on the object by processor 1 involves modifying Part B, then the local rep must locate and modify the single copy of Part B, synchronizing appropriately when necessary. The clustered object is identified by a single OID which is valid (in this address space) across the system and used by processors 1 and 2 in this example.*

cache thrashing due to write-shared state. In contrast, the file cache manager (FCM) for a particular region of the program's address space may be modified frequently, particularly if the program is experiencing a significant amount of paging or I/O. Therefore, replication is likely not a good choice for FCMs, since the cost of keeping the replicas consistent is likely to outweigh any benefits. Rather, it may make sense to partition the pages of an FCM across the set of processors accessing it,[1] or, in the case of a process' stack for example, to migrate the FCM as the process migrates.

The way different representatives of a clustered object communicate can also vary, primarily between data shipping and function shipping. The data shipping approach corresponds to the traditional shared memory approach of operating on remote data by directly making remote memory accesses (and thus, implicitly shipping the data from the memory to the cache of the processor operating on it). In contrast, function shipping corresponds to the traditional message passing approach of sending a message to the processor with the data and requesting it to operate on the data in some way. Both approaches can be appropriate depending on the desired tradeoffs between efficiency, scalability, and simplicity. Some issues that need to be considered when choosing between these two options include: the number and type (read/write) of memory accesses, the locking protocol used, the potential for high contention, the amount of sharing expected, and the expected cache hit

[1] The partitioning strategy might try to follow the logical partitioning of the data among the processes of the program, or just randomize the mappings in order to distribute the load evenly, depending on the access pattern.

rate. Many of these issues have analogues in the discussion in Chapter 2 on shared-memory multiprocessor performance issues.

The clustered object responsible for managing the clustered object subsystem itself is a good example of some of the alternatives and tradeoffs between data and function shipping. For example, a token passing scheme (used to coordinate clustered object destruction system-wide and discussed further in Section 4.2.4) is implemented by having the processor that has the token pass it on by simply writing a flag in the representative corresponding to the next processor in the chain. In this case, because there are no locking issues, and the token passing involves just a single store, remote memory access is a reasonable choice. In the case of clustered object cleanup, a message passing approach is more appropriate, since the key data structures are located on the home node and the cleanup operation can be quite complex.

In the following sections, we present the details of our clustered object infrastructure, beginning with the way clustered objects are referenced and followed by the data structures used to manage them. We also describe how local reps are created and deleted.

4.2.1 Object References and Translation

Although an OID is conceptually similar to a C++ reference, the OID must first be translated to a rep object reference before it can be used. The specific rep the OID refers to depends on the processor from which the reference is being made. It is important that the OID-to-rep translation occur in a way that is scalable and efficient.

We use a per-processor[2] object translation table, to perform the OID-to-rep translations. Each OID has an entry in the table identifying the local rep that should be used for the local processor. The clustered object of Figure 4.4 would have the table entry on processor 1 point to rep 1, while on processor 2 it would point to rep 2 (see Figure 4.5). To invoke a method on the clustered object, the OID is used to index into the table and the reference in that entry is used to invoke the corresponding method on the local rep.

Since each processor may have a different local rep for a given clustered object, each processor needs its own translation table. To avoid the need to explicitly locate the local table, each processor's table is located at the same virtual address so that an OID can be represented as a pointer into the table. Tornado supports a special memory region type specifically to allow this type of *aliased* mapping (where a given virtual address can refer to different data on different processors).

To allow each processor to allocate new OIDs without having to synchronize with other processors, the range of OIDs is partitioned and each processor allocates from its own subrange. This

[2]Throughout this dissertation the use of the term *processor* is often used as a shorthand for *virtual processor*, since in most cases there is a simple one-to-one mapping between virtual processors and physical processors.

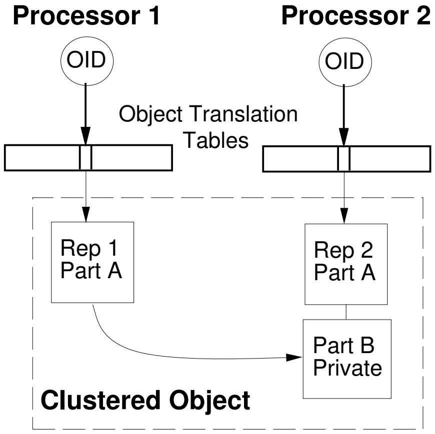


Figure 4.5: *Each processor has a private object translation table which is used to translate an OID to the local rep for the processor.*

also facilitates reuse of recently-released entries, thus improving spatial locality by having multiple valid entries on a cache line, and improving temporal locality by reusing the same cache lines. An added benefit of this partitioning is that the home processor of a clustered object (the processor where the clustered object was originally created and where certain bookkeeping information is stored) can be identified given its OID, as the OID remains the same for the lifetime of the object.

4.2.2 Translation Misses

From a space and time overhead point of view, it is not feasible to initialize all translation table entries on every processor, especially since most of them will never be used. Moreover, it is not possible to know *a priori* which processors will access a given clustered object. For this reason, translation tables are filled, and representatives are created, on demand when a processor invokes an operation on a clustered object for the first time. Since objects can have different policies for creating and sharing reps, it is necessary to invoke object-specific code when an uninitialized translation table entry is accessed. While translation misses are much less frequent than hits, it is still important that both the task of directing the miss to the object-specific code and the task of handling the miss are efficient and scalable.

An auxiliary global translation miss table is used to record a *miss handling object* (MHO) for each clustered object (see Figure 4.6). This table is partitioned across the system so that each processor maintains the portion corresponding to its range of OIDs. When a clustered object is created, it registers an MHO in the translation miss table. Upon a translation miss, the clustered object system examines the target OID to determine the entry in the global translation miss table that holds a pointer to the appropriate MHO, and calls the MHO to have it handle the miss. Typically, the MHO

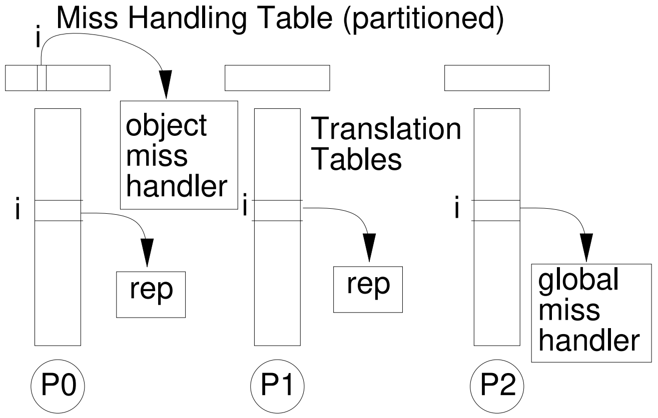


Figure 4.6: *This figure shows the state of the system for clustered object $\boldsymbol{i}$ after it has been accessed on processors $\boldsymbol{P0}$ and $\boldsymbol{P1}$, but has not yet been accessed on $\boldsymbol{P2}$. Reps have been installed on the first two processors, while the $\boldsymbol{P2}$ entry still points to the global (or generic) miss handler.*

will either create a new rep or locate a nearby existing rep, and fill in the appropriate translation table entry accordingly. On completion, the clustered object system will invoke the rep provided by the MHO, which handles the call as if it had been called directly.

The MHO may also choose to handle the call itself (or forward the call directly on to some other object), for example if no further calls are expected (e.g., destroy method) or if the policy is to only create local reps after a certain number of calls have been made. Because each clustered object can define its own MHO, different miss-handling policies are possible on an object-by-object basis (even for objects of the same type).

We use a simple scheme to avoid having to explicitly check for a translation miss on every access to an object. The object translation table entries are initialized as pointers to a special object that will handle the first part of the miss handling process (i.e., the part of looking up and invoking the object's MHO). This object has as many virtual methods as any of the clustered objects. When an object method is invoked for the first time on a given processor, the corresponding method in the generic miss handling object is invoked instead of the target clustered object's method. The generic MHO code locates and calls the MHO for that clustered object, as described above, which then (usually) replaces the object translation table entry with a pointer to the processor's local rep. The only drawback with this approach is the requirement that all clustered object methods be virtual. However, with our overall object-oriented strategy, this is generally required anyway.

To illustrate the different components involved, Figure 4.6 shows the state of the system for a fully replicated clustered object $\boldsymbol{i}$, after it has been accessed on processors $\boldsymbol{P0}$ and $\boldsymbol{P1}$, but not $\boldsymbol{P2}$. Because it has not yet been accessed on $\boldsymbol{P2}$, the translation table entry on that processor still points to the generic global miss handler. On a miss, the global miss handler will look up the miss handling

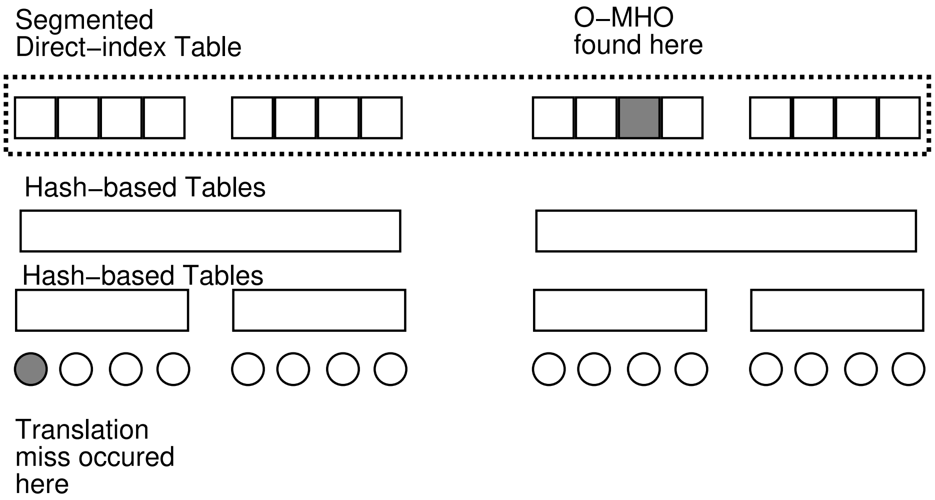


Figure 4.7: *This figure illustrates the hierarchy of global search tables. The top table is logically a single table but physically distributed across the system.*

object (MHO) for object i, which is located in the portion of the table held on $P0$. The global miss handler then forwards the miss request on to i's MHO for processing. When complete, a new rep will be created, and the entry in $P2$'s translation table will point to it.

Scalability

Although it may be sufficient to have a translation miss table that is partitioned (and not replicated) for most systems, for very large systems (of several hundred processors) creating all local reps from a single miss handling object may lead to unacceptably high contention. To deal with this problem, a more scalable infrastructure has been designed but not yet implemented. The scalable version calls a nearby rep to handle the miss rather than going to the central miss handling object in all cases. A hierarchy of hash tables are used to record the location of local reps that have been created in a subtree of processors below, as illustrated in Figure 4.7.[3] When a miss occurs, the hierarchy of tables is scanned, from bottom up, in order to find the closest rep to handle the miss. (Lower levels can be skipped if the miss is near the home of the clustered object.) Multiple concurrent misses can also be combined at the lower levels if necessary, further reducing the possibility of contention. Whether such a design is necessary—and at what system sizes and workload intensities—will require further experimentation with the current system.

[3] In NUMAchine, these subtrees would likely correspond to rings, but they do not necessarily have to.

4.2.3 Managing the translation table memory

Physical memory for the translation tables must be carefully managed, particularly in the kernel which normally never pages. There can be a large number of clustered objects (tens of thousands per processor) and the translation table on each processor has to be large enough to handle all objects created anywhere in the system. However, many clustered objects are only ever accessed on the processor on which they were created, and in general, it is expected that the sections of the translation table corresponding to remote processors will be progressively more sparse as the distance to the processors increases.[4] Given the size, sparseness, and locality of access of the translation tables, it makes sense to keep them in virtual memory, even in the kernel. Although the increase in the TLB miss rate could compromise performance, the locality in access pattern should keep the extra misses due to the indirection through the mapped translation table to a minimum. To keep the kernel simple, when memory runs low we simply discard victim pages rather than paging them out to disk, since the table is really just a cache of entries, with the miss handlers of the clustered objects keeping track of the existence and location of the reps (i.e., they maintain the backing copy). [5]

4.2.4 Destruction

There are a number of factors that complicate the destruction of clustered objects:

- To destroy a clustered object, one must coordinate the destruction of a collection of representatives spread over a potentially large number of processors.
- Multiple processes may be in the process of executing clustered object calls through any number of the clustered object's reps.
- While destroying a clustered object, there may be some processors that are handling misses and creating new representatives for the same clustered object.
- One must ensure that there are no dangling references to the clustered object that might be used in the future.
- Dangling rep pointers may also exist if destruction were to take place between the time the clustered object reference is dereferenced and the rep object pointer is used.

---

[4] Recall that the translation table is partitioned such that each processor only allocates objects from a certain range.

[5] As an optimization, it might make sense to compress the victim pages to a fixed size compression table (i.e., a second-level cache), because of the sparseness of the table. A simple compression scheme such as replacing consecutive null entries with a run-length would likely be sufficient. However, we have not yet implemented this.

An additional factor that further complicates the destruction process is the relationship between locking and object destruction. There are two kinds of locking issues in most systems: those related to concurrency control with respect to modifications of data structures, which we refer to simply as *locking*, and those related to protecting the existence of the data structures; i.e., providing *existence guarantees* to ensure the data structure containing the variable is not deallocated during an update. In a traditional system, before attempting to lock an object in which the lock variable is actually part of the object, it is necessary to obtain some guarantee that the object will not be destroyed between the time the reference to the object is obtained and the attempt is made to acquire the lock. This restriction is necessary since the act of locking the object involves modification to the object (namely setting the lock variable itself). If the memory of the object were reallocated to some other object during the attempt to lock the object, the lock acquisition would corrupt the new object. Furthermore there would be no way for the process acquiring the lock to know that the object had been destroyed and reallocated.

There are a number of ways of eliminating races between one process trying to lock (or in general, access) an object and another trying to destroy it, each with its own drawbacks. The traditional way is to ensure that all references to an object are protected by their own lock, and all the references are used only while holding the lock on the reference. The disadvantage of this approach is that the reference lock in turn needs its own protector lock, with the pattern repeating itself until some root object is reached that can never be deallocated. This results in a complex global lock hierarchy that must be strictly enforced to prevent deadlock, and it encourages holding locks for long periods of time while operations on referenced objects (and their referenced objects) are performed. For example, during a page fault, the traditional approach would require holding a lock on the Program object for the duration of the page fault, solely to preserve the continued existence of the regions it references.

Another approach is to use reference counts on objects, as was used in Mach [Black *et al.*, 1991]. With a reference counting system, a counter is kept in every object and is incremented whenever a new reference to the object is obtained, and decremented when a reference is released. Because destruction of an object is delayed while the reference count is non-zero, no locks need to be held to protect the object from being destroyed while a call is being made. However, cloning a reference can be an expensive operation, since it requires locking the object to increment the reference count and the same again to release it. Hence, it can actually increase the number of locking operations required if over-used, as well as increase the write traffic for the object.

A different approach altogether is to use lock-free concurrency control. However, practical algorithms often require additional instructions not currently found on modern processors, and they have their own difficulties in dealing with memory deallocation [Greenwald and Cheriton, 1996, Herlihy, 1993]. The tradeoffs involved in lock-free concurrency control will be more thoroughly considered in Chapter 6.

The Tornado approach

We address the issues of destruction in Tornado using a combination of features of the clustered object system and a novel semi-automatic garbage collection scheme. This allows a clustered object reference to be safely used at any time, whether any locks are held or not, even as the object is being deleted. This simplifies the locking protocol, often eliminating the need for a lock entirely (for example, for read-only objects). It also ensures that all locking issues can be contained within the object, increasing modularity and obviating the need for an inter-object locking protocol in most cases.

The key to our approach is to separate the clustered object existence guarantee issue (concerned with the validity of an OID reference prior to its use) from the locking issue (concerned with controlling concurrent accesses to data within an object). To address the guaranteed existence issue we distinguish between two types of references: what we call *temporary* object references and *persistent* object references. Temporary references are all object references that are held privately by a single process, such as references on a process' stack; these references are all implicitly destroyed when the process terminates. In contrast, persistent references are references stored in shared memory that can be accessed by other processes and can survive beyond the lifetime of a single process.

We use this distinction between persistent references and temporary references to divide destruction into three phases. In the first phase, the object that is to be destroyed ensures that all persistent references to it have been deleted. This is part of the normal cleanup process in any system when an object is deleted, as references to the object must be removed from lists and tables and other objects. Next, the object informs the clustered object system that it would like to be destroyed. This starts the second phase where the clustered object system insures that all temporary references have been eliminated. How this is achieved is described below. Finally, when no temporary references remain, the clustered object system calls back to the object to inform it that it is now safe for it to be fully destroyed (i.e., for all the reps to be destroyed, their memory released, and the clustered object ID freed).

The key problem is tracking the implicit destruction of temporary references. We begin by considering just the uniprocessor case and then move on to the multiprocessor case. Although there are many options (as covered in various garbage collection papers), we chose the one we felt was the simplest and most efficient for our purpose. It is based on the observation that system servers are event driven and that the processes that service those events have a short lifetime.[6] We keep a (per-

[6]For Unix-like systems, one can consider each system call or event handler as a separate process for the sake of this

processor) count of the number of active processes: every time a process is created (i.e., a call to the server from some external client process is received), the count is incremented, and when the call completes and the process terminates, the count is decremented. We can therefore be sure that if the count is zero, there can be no live temporary references to any object (on that processor).

Hence, when an object is to be destroyed and all persistent references have been removed, we simply wait until the count of active processes goes to zero, at which point it is safe to destroy the object. (Note that we are considering only the uniprocessor case at this point.) Of course, this is a much stronger requirement than actually required; for example, it would be sufficient to ensure that all processes that were active at the time object destruction was initiated have completed, but that would require keeping track of much more information than just a raw count.

One problem with our chosen approach is that there is no guarantee that the count of live processes will ever actually return to zero, which could lead to a form of starvation. However, since system server calls tend to be short, we do not believe this to be a problem.[7]

In order to efficiently determine when the count reaches zero, the active count has a special WAS_ZERO flag that is set when the count goes from non-zero to zero. This flag is reset by the clustered object system when it is given an object to destroy and checked periodically to see if the count has gone to zero. Although the counter adds to the critical path for all calls, it has good cache locality because it is shared by all calls, and the code is very short, particularly for the common case (see Figure 4.8). Because it is only necessary to track transitions to zero when a cleanup operation is pending (and even then, only the first such transition must be noted) the code is optimized for the case when no tracking is required. The key to the efficiency of the code in Figure 4.8 is that when tracking is required, on the first transition from one to zero the highest bit of the count is set. Thereafter, whenever the last active process leaves the system, the actual value in the active count variable will still have the high bit set and hence the code will fall through the test for "active equal to one", skipping the extra work. In the mean time, it is only necessary to check the highest bit of the count word to see if the count has gone to zero and reset it to turn on future checks. This keeps the common case code as short and straight as possible.

For the multiprocessor case, we need to consider all processes running on any processor that might contain temporary references to the clustered object being destroyed. Because of this, we need to extend the technique so that it waits until the active count goes to zero across all processors. Preferably we would like to bound the number of processors that need to be checked; however, this can be difficult: as soon as a reference is stored in shared memory, any process on any processor can

discussion. For operations involving I/O, we split the request into separate initiate and complete operations.

[7] One approach under consideration is to periodically swap the active count variable, so that the count of new calls is isolated from the count of previous calls. More careful investigation is still required however.

```
funct IncActiveCalls ≡
   x = &active;
   do
      y = load_linked(x);
      y = y + 1;
      z = store_conditional(y, x);
   while z = 0;
end

funct DecActiveCalls ≡
   x = &active;
   RETRY:
   do
      y = load_linked(x);
      if y = 1 then goto AT_ONE fi;                /* about to decrement to zero */
      z = y − 1;
      z = store_conditional(z, x);
   while z = 0;
   exit                                    /* common case of simple decrement is done */

   AT_ONE:                 /* active is 1, so we should store 0, but we want to set a flag */
   y = HIGH_BIT_SET;                  /* constant with 1 in highest bit and 0 elsewhere */
   z = store_conditional(y, x);
   if z = 0 then goto RETRY fi;                                /* retry from the top */
end
```

Figure 4.8: *Pseudo-code for incrementing and decrementing the count of active processes. Each statement is roughly one assembly instruction, and each variable represents a register.*

in theory read that reference into a local variable. However, in many cases, widely shared objects use a distributed structure that limits the access to a given representative to a small set of processors. Any reference stored in a representative can be accessed only by the set of processors that have already made an access to it, and because the first access to an object on a processor always results in a miss, each object knows the set of processors that can access any of its representatives. Hence, as part of cleaning up all persistent references, the clustered object being destroyed determines the set of all processors that could potentially have a reference to it, by forming the union of all the processors that have previously accessed any other object that has a persistent reference to it. This union is easily formed since each object already knows through the clustered object system which processors have accessed itself, and can communicate this information to the object being destroyed when that object requests that its persistent reference be removed.

Once one has collected the set of processors that potentially hold references to the object being destroyed, there is still the problem of figuring out when all the active counts on all of these processors has gone to zero. For this we use a token that circulates among the processors. When a processor receives the token it waits until its count of active processes goes to zero, before passing it on to the next processor. When the token comes back to the initiating processor it knows that the active count has gone to zero on all processors since it last had the token. At that point, any objects that were pending destruction when the token was last circulated can be told to cleanup.

To deal with scalability, there can be multiple tokens circulating, covering different subsets of processors. The set of processors involved in an object's destruction can then be used to determine which token to wait for.[8]

4.2.5 External Access to Clustered Objects

Up to now, we have described how clustered objects are invoked within a single address space. However, clustered objects are used throughout the system—in the kernel, servers, and user applications—and need to interact to provide their services. For example, opening a file requires communication between the clustered objects in the application (representing the open file state), the name server (to look up the absolute path), the file server (where the file is actually managed), and the memory manager (which sets up the mapped region for the client and brings data in from the file server). To make these cross-address-space interactions efficient, the same principles that are applied to the intra-address-space clustered object calls need to be applied to inter-address-space calls as well. In particular, calls should be directed at local representatives and should involve little or no shared memory accesses or locks to complete.

Cross-Address-Space Interactions

In order to be accessible from another address space, a target clustered object may require up to three additional support objects.

- An *interface* clustered object (one per target clustered object): this is the clustered object that external clients communicate with and that shields the details of the communication mechanism and authentication issues from the target clustered object. It accepts requests from external clients and forwards them to the target clustered object after appropriate security checks have been performed.

[8] Currently, all clustered objects are pessimistically assumed to be potentially accessible from any processor at the time of destruction, and hence only a single token covering the entire system is used.

- A *meta* clustered object (one per interface class): because some interface objects require **static** C++ methods for which there is no associated object, the meta object acts as the destination for external clients for such methods.

- A *proxy* object (one per client per interface object): this object resides in the address space of the client and shields the client from interprocess communication mechanism details, allowing the client to interact with a local object (the proxy object) as if it were interacting directly with the target interface object.

The relationship between these components is illustrated in Figure 4.9.

An important distinction between the target clustered object and the interface and meta clustered objects is that the target object can only be accessed from within the address space it is contained in and hence is called an *internal* clustered object, while the other two are *external* clustered objects that can be accessed from other address spaces as well as from within. The distinction is necessary for a number of reasons. First, an external object will, in general, need to authenticate the caller, while such authentication is unnecessary for intra-address-space calls. Second, an external object must interface with the interprocess communication facility. Finally, parameters need to be marshaled and demarshalled for cross-address-space calls.

The role of the interface object is to export an interface to clients that they can use to access the functionality of the internal object, while hiding details of the internal object that should not concern (and should not be accessible to) external clients. The interface object handles all of the server-end aspects of cross-address-space calls (like authenticating the caller, demarshalling arguments, etc.), and translates incoming requests into corresponding requests to the internal clustered object. Moving authentication and interprocess communication concerns out of the internal object makes the internal object more efficient for calls that come from within the address space.

The meta object and the proxy object are generated automatically from the interface object. The proxy object is a simple C++ object that allows the client to invoke methods on the interface object by calling the corresponding proxy methods. The proxy object marshals the arguments and invokes the appropriate interprocess communication call to transfer control to the corresponding interface object in the server's address space. As explained above, the sole purpose of the meta object is to provide a target for static C++ methods in the interface object. Static methods are used for class-specific operations that are not directed at any particular object, such as object creation. The proxy object directs static method calls to the corresponding meta object, which in turn invokes the static method of the interface object. This ensures that all cross-address-space calls are directed to a clustered object (which is required for security and authentication reasons).

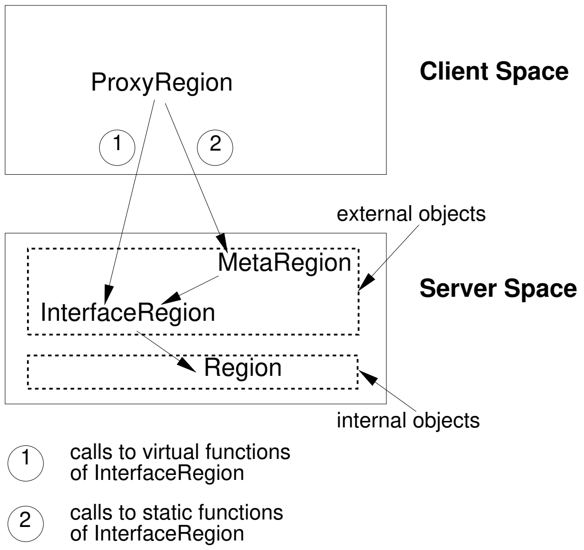


Figure 4.9: *This figure shows the relationship between the different objects used in client-server interactions for a generic Region object. The normal virtual method calls are handled by the interface object (InterfaceRegion) directly; the static method calls are handled by the external metaobject (MetaRegion) which simply redirects calls to static methods of the external object class.*

Implementation Details

Interprocess communication in Tornado takes the form of a protected procedure call (PPC, discussed in detail in Chapter 5). On startup, each Tornado server identifies to the kernel the entry point that will handle external requests to the server. This entry point is given a *PortID* by the kernel. The PortID, together with the clustered object ID (OID) of the interface object, uniquely identify an external clustered object in the system. Proxy objects contain these two pieces of information to identify the particular external object for which they are acting as a proxy.

When a client invokes a method of a proxy object, the proxy calls the PPC facility identifying the particular PortID which it wants to call and passing the OID and method number as two of the arguments to the server. The PPC facility directs the request to the specified server, where the entry point of the server performs some generic validation (described below) and invokes the specified method on the specified external object.

In order for the server to be able to ensure that the OID and method number it has been given are valid, four additional fields are provided in the clustered object translation table:

- A flag indicating whether this clustered object is externally accessible.
- A field containing the number of virtual methods defined for the clustered object. This is used to ensure that the method number passed as a parameter to the PPC call corresponds to a valid method in the external object.

- Check bits that are used to increase the period between object identifier reuse. The check bits are put in unused parts of the OID and also stored in the translation table entry. When a call is made, the check bits are removed from the OID and compared with the check bits stored in the table entry.

- An optional badge field which identifies a particular client program that can make requests to the external object. If this field is valid, the PPC-handling code verifies that the badge of the calling program matches the value stored before allowing the invocation to continue.

Given an OID and a method number as an argument to a PPC call, the server entry-point code performs the following sequence of operations: it verifies that the OID is in the range of valid object identifiers, is properly aligned, and corresponds to a valid external object; it then verifies that the method number is in the correct range and, if the badge field in the table is valid, verifies that the badge of the caller matches the stored value; finally it dereferences the pointer to the local rep recorded in the table entry and invokes the virtual method indexed by the method number.

Once the above checks have been passed and the interface method called, the next step is for the interface object to authenticate the caller. There are three primary options for performing authentication. The simplest case is when the interface object supports only a single client. Under these conditions, the client's badge can be stored in the translation table entry so that the authentication will be performed automatically by the basic sanity checks above. In the second case, there are potentially multiple clients that can legitimately call the object. In this case, no badge checks are performed in the entry point; instead the interface object checks the badge of the caller (supplied in the PPC call) against a list of valid clients. Finally, there is the case of first time calls, such as opening a file. In this case, the interface object uses traditional authentication techniques, such as checking the user id associated with the client against an access control list associated with the target internal object.

4.3 Examples

In our implementation, large-grain objects, like Program, Region, and FCM objects, are candidates for clustered objects, rather than smaller objects like linked lists.[9] In this section, we describe in a bit more detail how clustered objects can be used, based on existing implementations and some that are planned for the near future.

[9] Although all the main objects in Tornado are clustered objects, not all of them currently take advantage of all the features of the clustered object system. In particular, many of them are implemented using a single shared rep, either due to time constraints, or due to the fact that they are not performance-critical.

Dram manager

Because the Dram Manager is responsible for memory across the entire system it must be highly concurrent. To achieve this, it uses one representative (rep) per processor, with the reps created at boot time. Each rep manages a separate free list to maximize per-processor secondary cache reuse, and each set of reps in the same hardware cluster shares a single pool of all free memory in the cluster. Allocation requests are handled locally (either from the rep's free list, or from the cluster's pool) if possible. If no memory is free, then the request is forwarded to neighbouring reps. Deallocations are always forwarded to the rep from which the memory was allocated.

Clustered object manager

The Tornado clustered object manager manages the clustered object system itself. Because many clustered objects are created, used, and destroyed on just a single processor, the metadata associated with each clustered object is maintained local to the processor it is created on. There is therefore one clustered object manager rep per processor, each maintaining the information about the clustered objects created there. Operations on clustered objects whose home processors are remote use shared memory to access the information for simple operations, but use remote execution for more complex ones, like destruction.

Process descriptor

The process descriptor clustered object represents a single running process. Since the vast majority of the accesses to the process descriptor are from the processor on which the process is running, there is always only one rep and it is located on the same processor as the process. All other calls use the rep remotely (for information lookup) or forward their calls to the processor on which the rep resides for handling (such as for a wakeup call). If a process is migrated to another processor, then the rep is migrated along with it.[10]

Program

A more complicated example is the Program clustered object in Tornado. Because a program can have multiple processes running on multiple processors and most of the Program object accesses are read-only, the Program clustered object is replicated to each processor the program has processes running on. Some fields, like the base priority, are updated by sending all modifications to the home rep and broadcasting an update to all the other reps. Other components, like the list of memory

[10] Although part of the design, process migration is not currently implemented.

Operation	Instructions	Cycles
Call (hit)	6	
Call (miss)	150	206
Creation	250	465
Destruction	480	930
Stub overhead	10	
External call	49	

Table 4.1: *Time (in instructions and cycles on NUMAchine) for some basic operations of the clustered object system. Entries left blank are too short to measure effectively within their normal context on NUMAchine, but should not deviate too radically from their instruction count since they involve few memory accesses.*

regions in the program, are replicated on demand as each rep references a region in the list for the first time, reducing the cost of address space changes (i.e, adding and deleting regions) when regions are not widely shared.

4.4 Performance evaluation

In this section we first look at the cost of the various components that make up the clustered object system and then examine the performance implications of applying clustered objects to some typical data structures found in Tornado.

4.4.1 Component measurements

Table 4.1 presents performance measurements of some of the key components of the clustered object system. A *Call (hit)* is a clustered object method call that *hits* in the translation table (i.e., no miss handler is involved), while a *Call (miss)* is one that invokes a miss handler as part of the call. *Creation* and *Destruction* refer to creating and destroying a clustered object. The cost of cross-address space overhead is covered by *Stub overhead*, which includes the client side cost, and *External call*, which includes the server side cost (but not the cost of the interprocess call itself, which is covered later in Chapter 5).

The results are presented in instructions and, where possible, in cycles as measured on NUMAchine.[11] The difference between instruction counts and cycles can be due to a number of factors. Foremost are cache misses, which can have varying effects depending on whether the miss is satisfied by the secondary cache, local memory, remote memory, or some other processors cache. For

[11] Some measurements of very short sequences cannot be accurately made on the running machine within the context they normally operate in, without severely perturbing the results.

short, cache friendly sequences, other effects, such as pipeline delays due to branches or even cache hits, can increase costs anywhere from 10 to 100 percent.

The cost of a clustered object call that hits in the translation table is one instruction more than a regular C++ virtual function call, that is, 6 instead of 5 instructions, a relatively minor additional cost.[12] A translation miss involving a minimal handler requires 150 instructions and 206 cycles for the entire sequence.[13] Although this cost is non-negligible, it is still inexpensive enough to allow its use as a general purpose mechanism for triggering dynamic actions. As an example of a dynamic action, rather than simply inserting a rep into the table on a miss, the miss handler could keep track of the number of misses and only install (or perhaps migrate) a rep after a certain threshold has been reached. Alternatively, the handler might choose to install a rep only when certain methods are called, in the meantime forwarding all calls to some existing rep.

The creation and destruction of clustered objects is fairly complicated, and this is reflected in the relatively high costs for these two operations (465 and 930 cycles respectively). The creation of a clustered object includes the allocation of its miss handler and the allocation and initialization of a clustered object entry. The destruction of a clustered object (as measured for this test) includes a request for destruction sent by the clustered object being destroyed to the clustered object system, a garbage collection phase (involving only a single processor and triggered explicitly in this case), a callback to the object, and the freeing of the memory and the clustered object entry. Although these costs are significant, we feel that the various benefits that clustered objects provide make this a reasonable tradeoff, especially if the ratio of object calls to object creation is high, as we expect it to be. In particular, clustered objects tend to reduce overheads elsewhere in the system. For example, because of the garbage collection system, fewer locks are needed and special case checks to deal with the destruction of an object can be avoided.

The stub overhead presented in the table is for a simple cross-address space call with no C++ pass-by-reference arguments; each pass-by-reference argument would require an extra load and store (as required by the system calling conventions). The cost does not include code common to all cross-address space calls that must save and restore callee save registers.[14] Although the stub generator is relatively simple-minded, it produces reasonably efficient code by generating assembly code directly and taking advantage of the register-based parameter passing mechanism that the underlying interprocess communication system provides (more details are provided in Chapter 5).

The final entry in Table 4.1 is for the overheads on the server side of a cross-address space call. This includes validating the target object reference and method number, locating the target object,

[12]On the down side, the extra instruction is a dependent load and hence is on the critical path for the call sequence.
[13]The entry can also be pre-filled to avoid the cost of the miss, should the rep structure be known in advance.
[14]The total cost for cross-address-space calls will be presented later in Chapter 5.

performing some basic authentication, and invoking the target method. Given the necessity of these types of checks for almost all IPC systems, we consider the cost of 49 instructions acceptable. Since the code requires only a couple of references to a single cache line (the translation table), it should scale with processor speeds.

Although efficiency is important for Tornado, scalability is even more important. Figure 4.10 shows the times required for creation, destruction, and miss handling when the same operations are executed concurrently on 1 to 16 processors. The tests are run with one process per processor, repeatedly performing the same operation in a tight loop. The results presented are the averages for a large number of iterations across all processes involved in a given test. Figure 4.10(a) also includes range bars indicating the range of times for the different processes in the test. Although miss handling performs well, creation and destruction show some increase as the number of processors increase, and the range of times within each test varies quite widely. Figure 4.10(b) compares the performance of NUMAchine to SimOS on these two problematic cases. SimOS appears to match many of the quirks of the real hardware, and actually performs worse for larger number of processors.[15] This provides some confidence in the ability of SimOS to explain these results, and allows us to use SimOS to measure and vary a number of architectural components to determine the cause of the performance fluctuation. As it turns out, the cause here, as in many other cases as we shall see, is excessive cache conflicts in our direct-mapped cache. Different processors have data structures mapped to slightly different addresses, causing some processors to see many conflicts while others see none. If we run the same tests under SimOS configured with 4-way associative caches (as shown in Figure 4.10(c)), the variances disappear and the performance of these concurrent operations is flat and uniform from 1 to 16 processors, indicating good scalability.[16]

4.4.2 Sample clustered objects

We now consider the application of clustered objects to a few simple data structures to illustrate some of the cost/benefit tradeoffs.

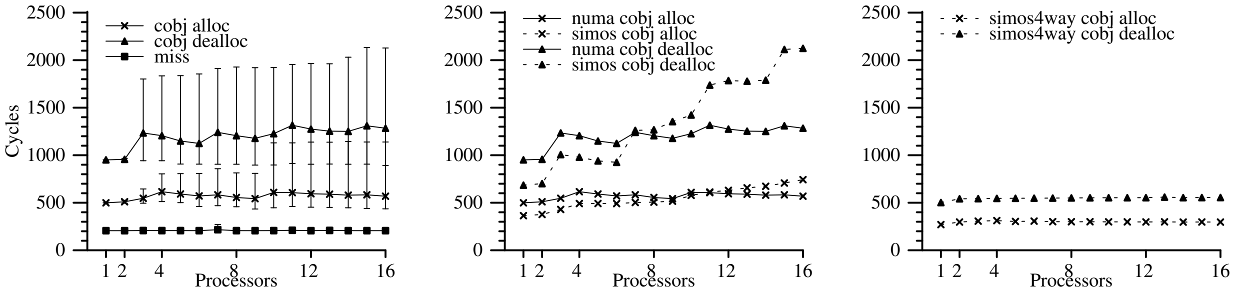
Performance counters

We begin with a very simple example that nevertheless illustrates many of the issues. We consider performance counters that simply count the number of occurrences of an event (or the total time spent under some condition).[17] Examples in existing systems include counters for context switches,

[15] As explained earlier, SimOS tends to over-estimate run times under high contention.

[16] Fortunately, most newer processors provide set-associative caches, so these pathological cases should become less common in the future.

[17] This example and the implementations are taken from the Master's thesis of Jonathan Appavoo [Appavoo, 1998].



a) Concurrent tests on NUMAchine b) Comparison of NUMAchine and SimOS c) SimOS with 4-way associative caches

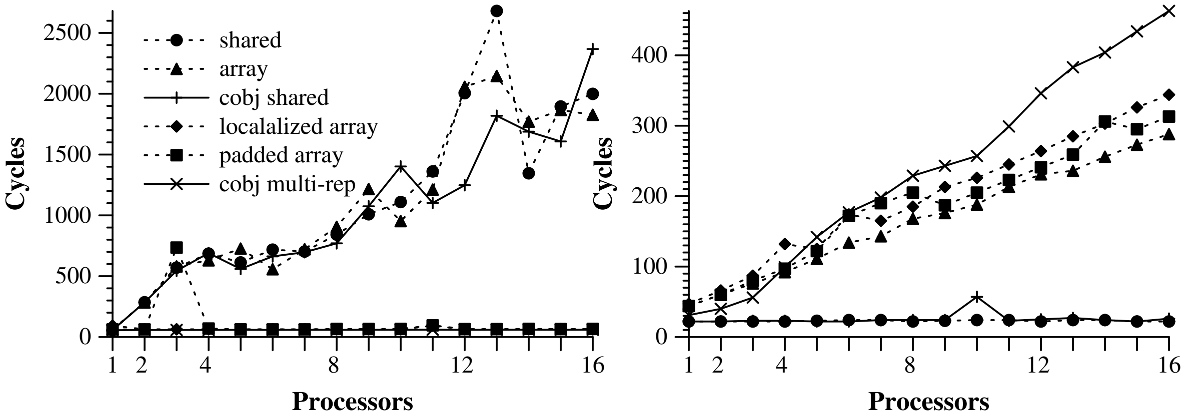
Figure 4.10: *These figures show results from concurrent stress tests on the clustered object system. The leftmost figure shows the time in cycles to perform a single concurrent allocation, deallocation, and clustered object call that misses. Each was measured by performing the test in a tight loop and dividing the resulting time by the number of iterations. The bars indicate the range of values measured across the different processes in the test. The middle figure compares the performance of NUMAchine to SimOS on two of the three tests. The final figure shows how the results change with a 4-way associative cache under SimOS.*

network packet receptions, or page faults. The nature of these types of performance counters is that they are incremented often (depending of course on the frequency of the event) but rarely read. Even when they are being actively monitored, generally only a subset of the counters are examined at a time, and usually at a rate of only once every few seconds, compared to an update rate of up to thousands of times a second. We use the nature of this access pattern to construct a number of different optimized versions of the performance counter.

As we saw in Chapter 2, the primary sources of performance degradation that we want to address are: (i) synchronization contention; (ii) cache coherence contention due to true or false sharing; (iii) memory contention; and (iv) remote access delays. Each of the different versions of the counter we will consider attempt to address different issues.

The results from tests of the different counter versions are presented in Figures 4.11(a) and (b). Each test in Figure 4.11(a) consists of running n processes concurrently, each repeatedly incrementing the same counter in a tight loop. The results are the averages across all iterations and all processes. The tests in Figure 4.11(b) are for a single process reading the value of the counter after the corresponding test presented in Figure 4.11(a). The details of the specific counter implementations and the implications of the results are discussed in the paragraphs that follow.

We first consider the non-clustered object versions of the counter, to get a sense of the issues to be addressed and what approach might be taken in the absence of clustered objects. The most basic counter is a single shared variable. For this, and all other versions, we use a lock-free sequence to atomically update the value without the need for a lock. Hence, the object consists of just an integer. Although lock-free, serialization still occurs in the atomic update sequence, and hence syn-



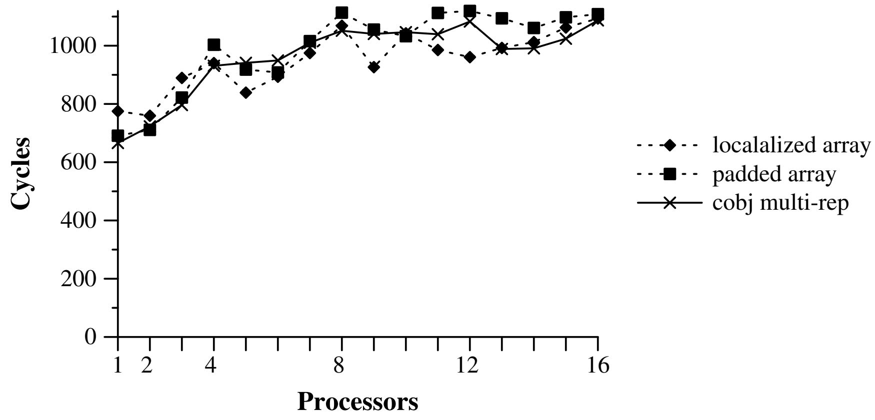
a) concurrent counter increment b) get counter value

Figure 4.11: *These figures show the results of two tests on a variety of different implementations of a counter. The results are presented in CPU cycles per iteration of the test. Figure (a) is for concurrent increments, while (b) is for a single processor reading the value of the counter. The different implementations are labeled as follows:* shared *is the straightforward shared counter variable;* array *is an array of counters, one per processors;* padded array *is the same as* array *but each element has been padded so that each is on a separate cache line;* localized array *is like* array *except that each entry points to a separately allocated counter (allocated from memory near its corresponding processor);* cobj shared *is the same as* shared *but implemented as a clustered object;* cobj multi-rep *is a clustered object version with one rep per processor maintaining a local count.*

chronization contention is a problem. This is clear from the *shared* case in Figure 4.11(a) which shows a linear increase in the cost to increment the counter as the number of contending processors increases.

To reduce synchronization contention we replace the single shared variable with an array of counters, with one entry per processor. Each processor then increments its own copy, which reduces synchronization contention. However, reading the value now requires reading each element of the array. This should be a reasonable tradeoff, given the access pattern described above. However, as shown by the line labeled *array* in Figure 4.11(a), performance is no better than the simple shared counter. This is due to false sharing: since the entire array fits within a single cache line, each write by a processor brings the entire cache line into its cache in exclusive mode, thus invalidating it from every other processor's cache, forcing them to re-retrieve their array element on the next increment.

To address the false sharing problem, we next try padding the array so that each entry occupies 128 bytes, or a full secondary cache line. This eliminates false sharing and provides the scalable performance we seek as shown by the line labeled *padded array*. However, it requires pre-allocating a large array and wasting most of the space due to padding. It also fails to provide any memory locality, since the array is allocated from a single memory module. This is not an issue with this test, since there is good cache locality, and the machine is relatively small. However, in larger systems



a) counter increment with cache flush

Figure 4.12: *This figure considers the performance of top three counters (for the increment test) when the counter is explicitly flushed from the cache before each increment. This gives some indication of their respective performance when the cache hit rate is poor.*

with less ideal cache behaviour, performance could suffer.

To attempt to improve locality, a final (non-clustered object) version is considered (*localized array* in Figure 4.11(a)), in which each element of the array no longer contains the counter, but instead contains a pointer to the counter that is allocated from memory local to the processor that will be incrementing it. As expected, there is no discernible improvement for this version because of the good cache hit rate for the previous version. To simulate the effect of a poor cache hit rate, Figure 4.12 presents results that include an explicit flushing of the counter from the cache before each increment. Although the differences are minor, the padded array version's performance is generally worse than the alternatives at larger system sizes.

We next consider clustered object versions of the performance counter. The first version is a direct translation of the simplest shared case to clustered object form. As expected, since it is still using a single counter variable with a single rep, the performance is roughly identical to the non-clustered object case (as shown by the line labeled *cobj shared* in Figure 4.11). However, if we switch to a fully replicated version with one rep per processor (*cobj multi-rep*), we get essentially the same performance as the *padded array* case. However, we also get the locality benefits of the localized array version, as seen in Figure 4.12. Furthermore, if we consider the base, uncontended costs as presented in Table 4.2, we see that the clustered object version is also noticeably more efficient than the array-based versions (56 cycles vs. between 62 and 93 cycles). This is due to the need for the array-based versions to determine the local processor number in order to look up the correct entry in the array (and the extra indirection in the case of the localized array version). Hence the clustered object version delivers both good contended and uncontended performance, with the additional benefit of providing a more structured environment.

Operation	Cycles
shared	55
array	62
padded array	62
localized array	93
cobj shared	57
cobj multi-rep	56

Table 4.2: *This table presents the uncontended costs for a single increment operation for the various counter implementations. All times are in cycles as measured on NUMAchine.*

However, there is a down side to the clustered object version, and all the versions that performed well on the increment test: the cost of reading the current value of the counter goes up significantly as the number of copies increases (see Figure 4.11(b)) because all values must be read to return the current value. As a result, the cost to read the counter is essentially the inverse of the cost to increment it, with the clustered object version performing worst of all. This demonstrates the expected tradeoffs that must be considered when choosing a structure for a variable such as this.

Hash table

A common data structure used throughout system software is the hash table. In Tornado there are many cases where a form of locality occurs naturally in the accesses to elements of the hash table. For example, the dynamic memory allocator (described in more detail in Chapter 6) uses a hash table in the kernel to map from memory block addresses to page descriptors. Because the allocator is geared towards maximizing NUMA locality, most lookups are for local memory (memory allocated from the local station). Similarly, a hash table is used to keep track of waiting processes for our bit-based locks (also described in more detail in Chapter 6), and there is natural locality between the processes that access a common lock.

In both these cases, and others, there is a natural way of partitioning the hash table so that operations on it are most often directed at the local partition. In the case of the memory allocator, the address of the block of memory already contains information about its home node (due to the design of the allocator). The same is true for locks, since the address of the lock is used as one of the keys.

Based on these examples, we consider two implementations of a hash table optimized for the types of uses just described: a single shared non-clustered object hash table, and a partitioned clustered object hash table. Both implementations use the same total amount of space (summing across all partitions in the case of the clustered object version), which is scaled according to the number of processors in the system. We assume some bits in the upper part of the key is used to identify the

Operation	Cycles
shared	325
clustered object	345

Table 4.3: *Base times per hash table lookup for a single shared hash table and a clustered object hash table.*

home node in the case of the partitioned hash table. (For our examples this assumption is reasonable, given that the key is just the address of some block of memory or a lock.) For the experiments, we prefill the hash tables with the same evenly distributed values, with an average of four elements per bucket. All experiments consist of n processes searching for a large number of uniformly distributed keys. To keep the tests fair, each process uses a distinct range of keys, none of which are in the hash table. This ensures that none of the processes benefit from the distribution of keys in the hash chains, since all searches fail requiring the entire hash chain to be searched.

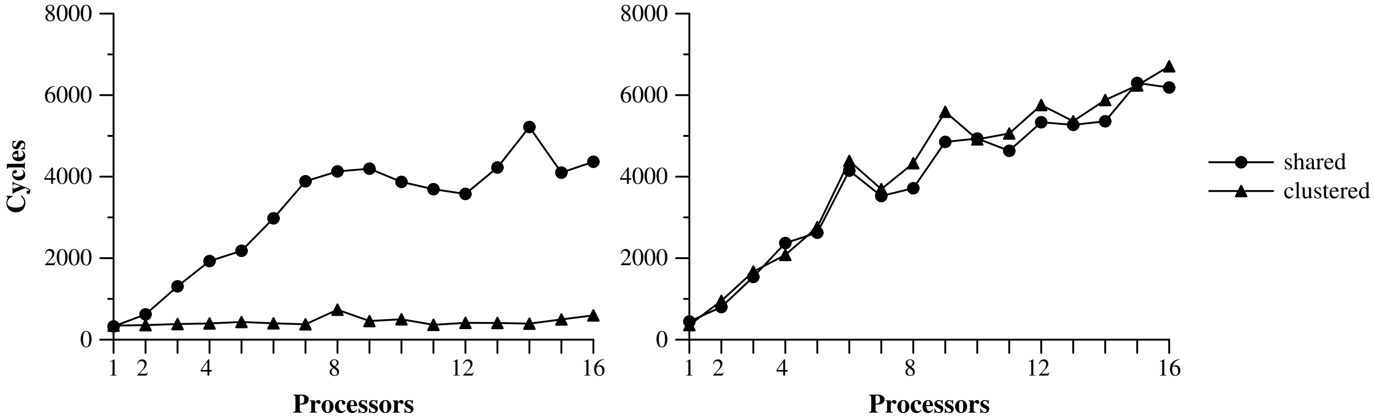
Table 4.3 shows the base, uniprocessor times, per lookup. The cost of the clustered object version (345 cycles) is slightly higher that the non-clustered object (*shared*) version (325 cycles), in part due to clustered object overhead itself, and in part due to the implicit cost of requiring the use of virtual functions. However, the difference in cost is still quite low, particularly when we consider multiprocessor performance.

Figure 4.13(a) shows the performance for a series of concurrent localized lookups, where each process looks up keys that are local to it. The performance benefits of partitioning the hash table are clear, as is expected given the locality of the requests. This is despite the fact that in the shared case, the requests are evenly distributed across all buckets of the hash table and the size of the table is grown with the number of processors. This is a result of the extra memory locality and smaller per-processor working set size of the partitioned hash table for this workload. For the case of randomly distributed requests, as shown in Figure 4.13(b), there is, however, no advantage in the clustered object version, but it does not perform significantly worse either.

List

Another data structure that is commonly used is the list. Lists have many different types of access patterns, but the type most applicable to clustered object optimization is a (shared) read-mostly list. Examples from Tornado include the list of virtual processors in a program, the list of badges allowed to access a file, and the list of memory regions in an address space. These are all lists that might be widely shared (at least for parallel programs) and that tend to be searched much more frequently than they are modified. As such, they are the perfect candidate for replication.

In Tornado, the Program clustered object replicates the region list to each processor the program



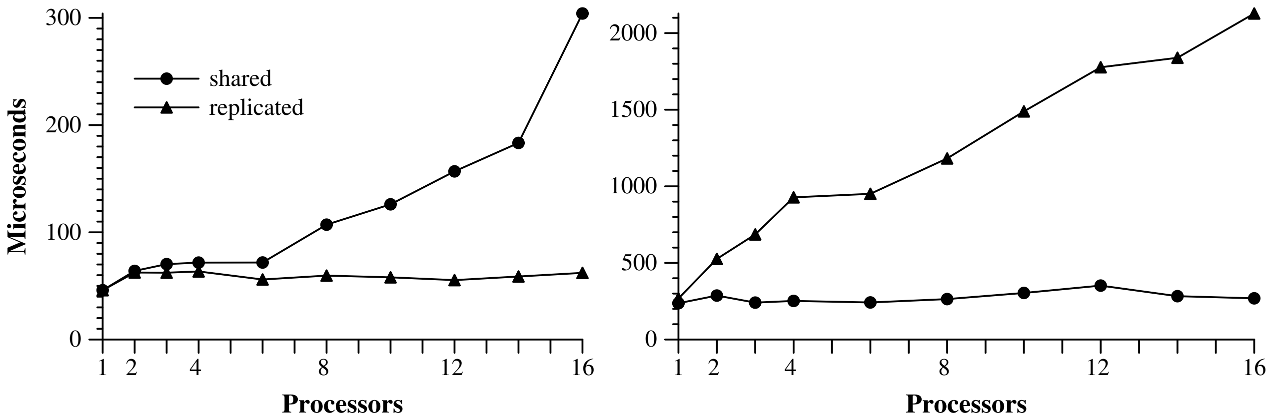
a) concurrent local hash lookup test b) concurrent random hash lookup test

Figure 4.13: *Figure (a) compares the performance of a single hash table to a partitioned hash table for a localized lookup. Like the other microbenchmarks, these results were obtained by measuring the time for a large number of lookups and dividing the resulting time by the number of iterations. Figure (b) compares the same two hash tables but for a completely random (non-localized) distribution of requests.*

has processes on. The replication happens on demand, as a process accesses a region for the first time. When regions are removed, an invalidation must be sent to all replicas. This is done from the home node (where the program started) in order to ensure proper serialization, so removal requests from other processors must be forwarded there. Similarly, additions to the region list are always forwarded to the home node and then propagated on demand.

The performance impact of the replication can be seen in a simple experiment where separate processes of a parallel program fault on independent regions concurrently. The data has already been paged in, so no I/O is necessary. However, in order to verify that the faulting address is valid and to determine what FCM it belongs to, the list of memory regions must be searched. The list needs to be locked for this operation creating a potential bottleneck.

Figure 4.14(a) shows the effect of having a single shared list versus a replicated list for page faults. Similar to the other experiments, the results are obtained by having n processes concurrently fault on a series of pages and then averaging the time over the total number of faults and the total number of processes. As expected, in the replicated case the cost stays relatively flat compared with the shared case, where the region list lock is clearly a bottleneck. However, as is often the case with these types of data structures, there is a tradeoff. Figure 4.14(b) shows the time it takes for a single process to delete a region for both cases. Here we see a reversal of the faulting case: as the number of processors grows, the cost to remove a region grows linearly, since all lists must be updated. The impact is highly exaggerated in this example, since this clustered object happens to use function shipping, which is quite expensive for this simple operation. However, under normal



a) concurrent page faults b) region deletion

Figure 4.14: *Figure (a) compares a single shared region list to a replicated region list for a test involving concurrent processes of a parallel program faulting on different regions. Figure (b) compares the same two region lists, but on a region removal operation.*

circumstances, the number of faults is generally expected to be much higher than the number of modifications to the region list, making this optimization worthwhile.

4.5 Open Issues

The work presented here represents just the beginning of the investigation into designing and implementing clustered objects. Most of the effort has gone into the clustered object infrastructure. Many issues remain to be addressed, primarily in the use of clustered objects on a large scale, but also in some areas of the infrastructure. Some of the more important open issues include the following:

- The size of the per-processor translation tables is still a potential problem. Ensuring that the virtual space is large enough to accommodate any reasonable (large) number of clustered objects on a single processor, particularly on a large system, could be a problem. In addition, the amount of physical memory needed to keep a reasonable working set of clustered objects in a large system, with very sparsely filled tables, is unclear.

- It still remains to determine whether there is a need for a method to ensure that garbage collection is not indefinitely postponed while waiting for an active count to go to zero. This can affect the entire system since it could hold up the circulation of the token.

- Scalability concerns for the garbage collection scheme remain to be investigated to determine whether there is a need for more than one circulating token.

- The migration of a clustered object rep is actually trickier than would first appear. In particular, if one tries to implement migration by first replicating the rep to the target processor and then removing the previous copy, then one is faced with all the complications of destruction; in particular, ensuring that no process is currently referencing that rep before it is deleted. The similarity to destruction indicates some possible avenues for attack (e.g., waiting for the active count to reach zero), but since the clustered object is not being destroyed, persistent references remain and accesses can continue at any time. This requires more synchronization in the object to deal with these problems, but not all the details have been worked out yet. For now, migration is performed by migrating state and roles, rather than the actual rep object.

- Managing distributed state, even if it is constrained to within a clustered object, is still a difficult task. It seems likely that higher level support services that facilitate managing such distributed state are needed. It may be possible to build standard distributed data structures using techniques similar to those used for the C++ standard template library.

4.6 Related work

The basic idea of improving performance for contended data structures through some form of distribution has recently become quite common. Examples can be found in most low-level system components developed recently, such as distributed shared memory systems [Scales *et al.*, 1996], performance monitoring systems [Anderson *et al.*, 1997], and dynamic memory allocation systems [McKenney and Slingwine, 1993]. The clustered object system goes beyond what these systems do by providing a general framework to facilitate the application of these types of optimizations.

Concepts similar to clustered objects have appeared in a number of distributed systems, most notably in Globe [Homburg *et al.*, 1995] and SOS [Makpangou *et al.*, 1994]. In these two cases and Tornado the goal is to hide the distributed nature of the objects from the users of the objects while improving performance over a more naive centralized approach. However, the issues faced in a tightly coupled shared-memory multiprocessor are very different from those of a distributed environment. For example, communication is cheaper, efficiency (time and space) is of greater concern, direct sharing is possible, and the failure modes are simpler. Hence, the Tornado clustered object system is geared more strongly towards maximizing performance and reducing complexity than the other systems.

Naturally, there are strong similarities between clustered objects and the clustered design of Hurricane [Unrau *et al.*, 1995] and Hive [Chapin *et al.*, 1995b], as previously discussed. The key difference is the application of clustering on an object-by-object basis, improving modularity and flexibility.

One key feature of clustered objects that is common to many systems is the extra level of indirection in the translation table. For example, many early Smalltalk systems used an object table to support paging of objects or to support garbage collection in a distributed system [Bennett, 1987]. Another more recent example is the use of an extra level of indirection in object references to support multiple versions of a given class within the same program, with dynamic class updates [Hjalmtysson and Gray, 1998]. Although their end-goal is quite different from ours, their basic infrastructure bears some resemblance to that of clustered objects, particular in the use of virtual functions in C++ to intercept calls transparently.

The clustered object destruction scheme has obvious parallels with garbage collection schemes [Wilson, 1992], particularly with multiprocessor and distributed garbage collection systems [Herlihy and Moss, 1991, Fessant *et al.*, 1998] which are similarly concerned with reducing communication between the processors. Although our garbage collection scheme is in some sense a hack, it works reasonably well in our environment. The particular algorithm bears some similarity to deferred reference counting garbage collection schemes in their attempt to avoid the need to scan stacks for references [Wilson, 1992]. It is also quite similar to the scheme described in IBM's patent 4809168, which also delays sensitive operations until all active processes have reached a known safe point. However, the IBM scheme appears to target uniprocessors only and is less general than ours.

4.7 Summary

The clustered object system provides the infrastructure to facilitate various important multiprocessor optimizations, such as localizing data structures and locks. The key to the system is the use of the per-processor translation table that allows a common clustered object reference to be used uniformly throughout the system while always directing the request to the most appropriate representative. It also provides a mechanism for uniformly referring to and accessing clustered objects across the various servers and applications, creating a single clustered object space for accessing all services.

Although there are tradeoffs in the cost and benefits of partitioning and distributing an object, these decisions can be made on an object-by-object basis and periodically revisited and enhanced as needed without altering the fundamental structure of the system.

Chapter 5

Protected Procedure Call

In the previous chapter we examined the internal structure and external interfaces of the Tornado clustered object system. In this chapter we look at the interprocess communication subsystem that allows clustered objects in different address spaces to invoke one another and that allows different representatives of a single clustered object to communicate across processors. We examine the particular aspects of Tornado and its target multiprocessor platform that influenced the unique design of the IPC system, and present a detailed implementation description as well as performance results.

5.1 Motivation

In a microkernel system like Tornado that relies heavily on client-server communication, it is important to extend the locality and concurrency of the architecture beyond the internal design of individual components to encompass the full end-to-end system. All application interactions with the kernel and system servers—to read a file, to draw on the screen, to get the time—require interprocess communication (IPC). In order to ensure the microkernel structure does not add additional overhead to an operating system design, the cost of these interactions must be similar to the cost of a system call in more traditional (Unix-like) systems. In addition, on a multiprocessor, the operating system must match the degree of concurrency in its internal structure with that of the hardware and the applications' demands. This requires the IPC facility to allow servers to handle multiple requests concurrently, ideally as many as there are clients. Finally, a shared-memory multiprocessor requires system software to maximize memory locality in order to achieve optimal performance. This means servers need to control where their processes run with respect to the clients they are servicing and with respect to the data used to satisfy those requests.

We therefore require an IPC facility that provides low latency, high concurrency, and high locality, and supplies the infrastructure that enables all system servers to be built with these same characteristics. In addition, these characteristics must be maintained independent of the type of workload, whether it consists of large numbers of independent sequential programs requesting services from independent servers, small numbers of large-scale parallel programs requesting services from a common server, or anything in between.

While designing Tornado's IPC facility (originally developed in the context of the previous operating system, Hurricane) we realized that the need for locality and concurrency required a new model of IPC, rather than just an improved implementation of a traditional message-based IPC subsystem.[1] We found that direct translation of uniprocessor IPC facilities to multiprocessors generally results in accesses to shared data and locks along the critical path. These shared data accesses can result in cache misses and increased cache invalidation traffic, adding hundreds of cycles to the cost of an operation, especially as the concurrency in the system grows. As the efficiency of IPC implementations increases, locks quickly saturate, even if the critical sections are very short. More importantly, the traditional IPC model, based on the notion of a single (or small number) server process that waits for requests to process, does not provide the natural concurrency and locality that is required if all parts of the system (and not just the kernel) are to scale effectively.

The new model we developed is based on the Protected Procedure Call (PPC) model in which a process is considered to cross protection domains to make a procedure call in another address space. The PPC model facilitates the implementation of an IPC facility that requires no locks or accesses to shared memory for a typical call. With PPCs, we were not only able to achieve the high performance we sought, but also found it straightforward and natural to extend it to support other types of control transfers that appear throughout the system, such as: (i) asynchronous requests (where the requester does not block waiting for the call to complete), (ii) interrupt dispatching, (iii) upcalls, and (iv) cross-processor calls.

5.2 PPC model

The Tornado PPC facility serves two main purposes:

- It provides a cross-address space communication facility that allows a process in one address space to invoke a method of an external clustered object in another address space.
- It provides a facility to allow a representative on one processor to invoke methods of another representative of the same clustered object on another processor.

[1] An example of the more traditional message-based IPC can be found in Hurricane's original facility based on the send/receive/reply model where messages are sent between processes; this in turn was derived from the V operating system's IPC facility [Cheriton, 1988].

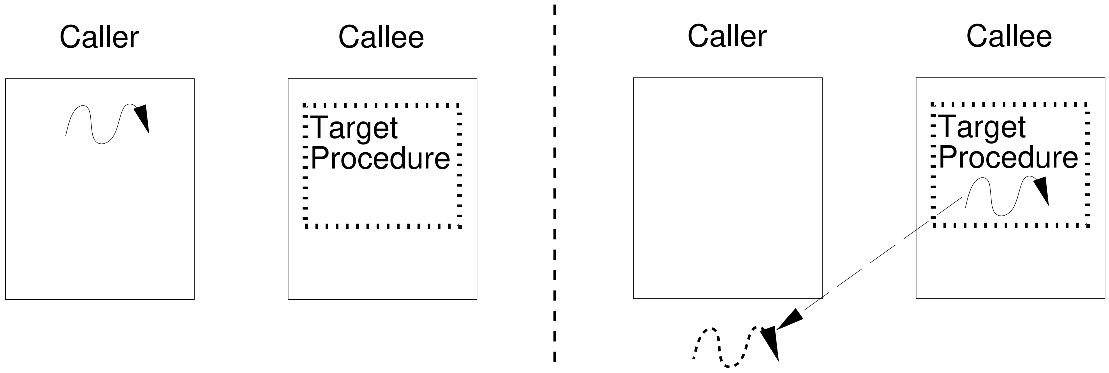


Figure 5.1: *The leftmost figure shows the state before the call. The process running in the caller address space (indicated by a wavy arrow) makes a call to the target procedure in the callee's address space. The rightmost figure shows the state during the call: the caller process is suspended and a new process in the callee address space is started with a link back to the calling process. When the call completes, the state will return back to the figure on the left.*

Hence the Tornado PPC facility provides both a communication facility that crosses domains while generally remaining on the same processor, and a facility that remains within a single domain while crossing processors. Of course, other combinations of these alternatives are possible (cross-address space, cross-processor calls, for example), but the PPC facility was optimized for these two uses. We describe same-processor, cross address space communication first and then remote processor communication later.

5.2.1 Overview of PPC

The basic model of a PPC is, as the acronym suggests, a procedure call that crosses protection domains. As such, it is by default a synchronous call, where the caller blocks until the callee returns. Also, the model suggests that the call is not directed at some waiting process, but at the procedure in the other protection domain (i.e., address space). One possible implementation would be to strictly follow the procedure call model and have the calling process actually switch into the protection domain of the callee, execute the code of the procedure, and then return back to the protection domain of the caller. However, for a variety of pragmatic reasons, we chose to implement the PPC by creating a new process in the callee protection domain to handle each call.[2] Hence a PPC call starts a new process in the target domain and blocks the caller; the callee process then executes the code associated with the call, and does a special return which terminates itself (the callee process) and unblocks the waiting caller (see Figure 5.1).

[2]How this is made efficient is described later.

The key advantages of the PPC model are that: (i) client requests are always serviced on their local processor; (ii) client and server share the processor in a manner similar to handoff scheduling [Black, 1990]; and (iii) there are as many threads of control in the server as client requests. Points (ii) and (iii) aid concurrency, while (i) aids locality by allowing servers to maintain client-specific state local to the client, reducing unnecessary cache traffic. For example, for page faults to memory mapped files that require I/O, all of the state concerning the file that the client has mapped can be maintained in data structures local to the process that is accessing the mapped file. In some sense, the PPC facility is similar to extending the Unix trap-to-kernel process model to all servers (and the kernel for Tornado), but without needing to dedicate resources to each client, as all clients use the same server port to communicate with a given server. The PPC model is thus a key component in enabling locality and concurrency within servers.

5.2.2 Parameter passing mechanisms

There are three ways of passing parameters back and forth through a PPC call (much of which is hidden from the user through the use of the proxy generation facility). First, the PPC facility allows up to eight register-sized arguments to be passed from caller to callee, and back again. This is done very efficiently by simply not using the eight registers holding the arguments while processing a PPC call in the kernel.

The second option is to use a special buffer provided by the PPC facility for passing page-size chunks of data between caller and callee. This facility provides two special regions of memory: one for an *OUT* page and one for an *IN* page. The OUT page is used to pass data *out* of the caller's domain and into the callee's domain. The data is received by the callee in its IN page. On return, anything that the callee places in the IN page is returned back to the caller's OUT page (see Figure 5.2). The implementation actually remaps the physical memory from one domain to another, removing the need for a copy. This also has the side effect of removing access to the caller's OUT page during the call, and the callee's IN page after the call. If a callee in turn needs to make a PPC call, it can fill in its own OUT page and pass it to its target and receive a response without interfering with the contents of the IN page. In addition, for the case where the same data is passed through several call chains, the IN and OUT pages of a process can be swapped, making the contents of the IN page the OUT page, and the OUT page the IN page.

Finally, for much larger quantities of data, the preferred method of data transfer is to establish a region of shared memory between the two protection domains. For example, a pipe might be created by establishing a shared buffer into which the producer would place data and the consumer would receive data, using the shared memory for synchronization, possibly obviating the need for PPC calls at all, except to establish the shared buffers in the first place.

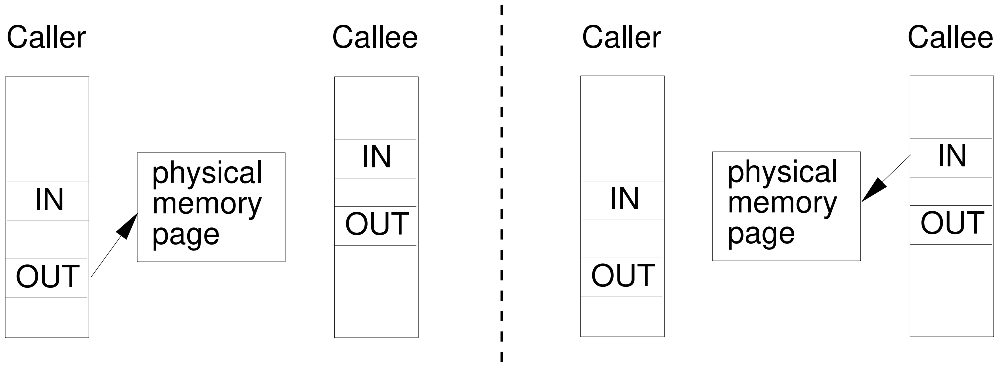


Figure 5.2: *The leftmost figure shows the state before the call. In the caller's address space, the OUT page region points to a physical page backing that region. When the callee is invoked the OUT page is removed from the caller's address space, and mapped into the IN page region of the callee, as shown in the rightmost figure. On return, the page is returned back to the caller's OUT page region.*

5.2.3 Naming and authentication

A client wishing to make a call to a server directs the request to the server's *port*. It uses a *PortID* to identify the port, which includes both a port number and either a virtual processor number or a special ANY_VP value. The latter is the common case, and normally results in the call being handled on the local processor. The parameters to the PPC call generally include the target object, the target method number, and parameters to the method. The entry point code in the server examines these parameters to determine which object and method to call.

PortIDs are generally hidden in proxy objects, together with the target external object's clustered object reference. Typical client code only interacts with proxy objects and not with PortIDs, external clustered object references, method numbers, or any other part of the lower layers of the PPC facility.

As opposed to systems like Mach [Accetta *et al.*, 1986] or Spring [Mitchell *et al.*, 1994] that use capabilities for both naming and security, we specifically chose to separate the two issues. Callers are identified to servers by their badge, which, as described earlier, is an identifier for the program or address space of the caller. The badge can be used by the server to retrieve client-specific state so it can verify whether the client is permitted to make the call. This separation of naming and authentication allows the service provider to choose the type and degree of security and protection it desires, allowing for server-specific optimizations. For example, as we saw in the previous chapter, when there is only a single client of a given clustered object, a direct check for matching badge can be performed as part of the external call processing code. With more than one client, a list of pre-authenticated badges can be used, or, in the case of a large number of clients, a hash table could be used.

The use of capabilities can have strong negative performance implications for two primary reasons. First, it can entail locking in the common path of an IPC call in order to verify that the capabilities are valid. Second, because capabilities are often added and deleted with great frequency,[3] the data structures must either be shared, incurring a high cost due to coherence traffic, or they must be replicated, with the attendant cost of keeping the frequently changing replicas consistent. This is not a problem for PPCs, since the global PPC data structures change only when servers are created and destroyed, which happens much less frequently. Avoiding capabilities thus facilitates the implementation of a highly concurrent and localized IPC subsystem.

5.2.4 Extended application of PPCs

In addition to the normal use of PPCs for protected procedure calls, PPCs are used in a number of other circumstances. First, an asynchronous version of PPCs is provided, where the caller does not wait for a response from the callee; it returns immediately and the caller and callee proceed concurrently. This is used in a number of cases, but its most prominent use is to start new processes in the same or different address space. In fact, all processes, even the first process in a program, arise due to a PPC call (either synchronous or asynchronous), as PPC calls are the only way to create processes in Tornado.

Second, interrupt dispatching is integrated into the PPC facility, using a technique similar to that used for asynchronous requests. An asynchronous request from the kernel to the device server is manufactured by the interrupt handler and dispatched as for a normal call. From the device server's point of view, the interrupt appears as a normal asynchronous PPC request.

Upcalls are essentially software-based interrupts and use the same implementation as the interrupt dispatcher, but may be triggered by an arbitrary system event, rather that just an interrupt. They have wide application and are currently used for debugging (breakpoints are converted into upcalls to the parallel debugger), signals, page faults, and exception handling (various traps may be converted into upcalls to the offending application).

All these situations benefit from PPC's ability to (i) bypass the general scheduling facility, (ii) maximize locality, (iii) dynamically create new processes, and (iv) support unconstrained concurrency. By comparison, in Hurricane, which used a more traditional IPC system, we needed a variety of special case solutions for each of these situations that were often significantly more complex and costly than the PPC-based approach.

[3] For example, in Mach *send* capabilities are often added and deleted with each request to allow the server to send a response back to the caller.

5.2.5 Remote PPC

So far we have only discussed PPCs in the case where the caller and callee (the process created on behalf of the caller's request) are on the same processor. A PPC can also be directed at a particular virtual processor for a given port. This form is primarily used for communication among representatives of a clustered object (hence, within a single address space), or to communicate with servers that handle physical devices that must be accessed from specific processors. In addition, asynchronous remote PPCs are used to create processes on different processors for parallel programs. A PPC request can be directed at a particular remote processor through two mechanisms: either the caller can specify the virtual processor as part of the PortID, or the server can specify the target virtual processor for all calls when the port is first created. Apart from potentially specifying a target virtual processor, there is no difference from the caller or callee's point of view between a local and remote PPC (although there is a difference in implementation as described below).

A natural extension of the remote PPC is a multicast PPC. There are numerous potential applications of such a facility, such as for sending updates to multiple representatives of a clustered object or for starting all the processes in a parallel program. However, there were enough unresolved issues surrounding both the implementation of such a facility and the interface to present to the user, that multicast PPCs have not yet been either designed or implemented. Such seemingly simple questions, such as how to report errors when some subset of the multicasted targets fails, how to collect multiple responses, and how to deal with IN/OUT pages, proved more difficult than initially expected.

5.2.6 Error handling

Error handling during a PPC transaction is relatively straightforward from a client's point of view. All PPC calls return a standard error code. The returned value can be set either by the server process executing the call, or by the PPC subsystem itself if it can't complete the call; standard error codes ensure that there is no confusion between an error in the PPC subsystem and one in the server handling the call.

If a server process executing a call is terminated for some reason, then its caller is unblocked and given a PPC return code to indicate the failure. If the server process was itself blocked on a PPC call to some other server, the server process being terminated is detached from its target, as if the call had originally been an asynchronous call, and the target continues until it returns, at which point it is treated just as if it was returning after an asynchronous call. If a server process needs to know if a client has died, it can check for the existence of a caller at any time.

5.2.7 Process Initialization

We have found that in some cases processes need to execute initialization code when they are first created (e.g., registering themselves with an exception server, or allocating a buffer). However, as we shall see in the next section, processes are not created anew on every call, but instead are recycled for the next call when they return. Therefore, this type of initialization only needs to be performed once for each newly created process. In order to prevent this one-time cost from impacting subsequent calls, we allow processes to change their own call handling routine independently. Thus, when a service entry point is created, the call handling routine specified is the initialization routine. The first call to a newly created process enters at the initialization routine which then, after initialization, changes the process' call handling routine to the normal handler so that subsequent calls bypass the initialization routine.

This option is currently used to initialize a few key structures for each process. This includes a per-process structure that allows each process to efficiently locate per-process information, such as its virtual processor number, its own process proxy, and the location of its IN and OUT pages. A special structure used by blocking locks for enqueuing the process onto a lock is also initialized.

5.3 Implementation

To present the Tornado implementation of PPCs, we begin with a brief overview of the steps followed for a common-case PPC call and return. We then retrace these steps more carefully, examining in more detail the different components that go into a PPC call and the various exceptional conditions that must be dealt with.

5.3.1 Overview

The common case PPC call path is quite straightforward. First, the client prepares in specific registers all the information to be passed to the PPC trap handler, including the type of call (e.g., synchronous or asynchronous), PortID, and the 8 register arguments for the server. The client is also responsible for saving on its stack all other registers that must be preserved across the call. Next, the client issues a trap instruction, which transfers control to the kernel as an exception. The kernel then saves some information, such as the caller's instruction pointer and stack pointer, in the caller's process descriptor.

At this point, the kernel is still in what we call "exception mode", which in Tornado means that it cannot be preempted and cannot acquire locks or access any virtual memory. Because of this, and to reduce the overhead of handling calls, all resources needed to complete a PPC call (for the common

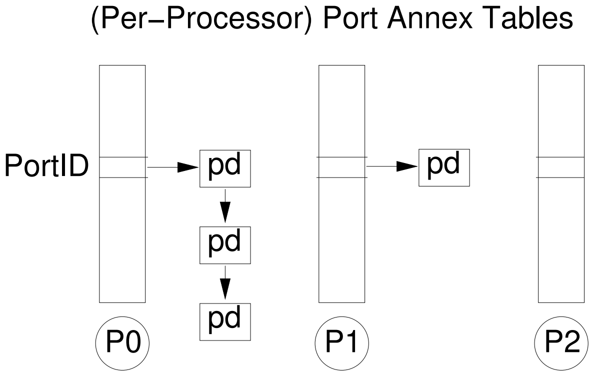


Figure 5.3: *Each processor has its own table of PortAnnexes indexed by the PortID. Each entry contains a list of PDAnnex descriptors (PDs) to use on this processor for calls to the port.*

case) are replicated on each processor. Hence, the call processing code can safely access any PPC data structure knowing it is protected by the fact that interrupts are disabled and only the local processor is allowed to access it. Coordination between different processors is handled by higher-level code that deals with creation, destruction, and various exceptional conditions (described later).

The next step in processing a call is to use the PortID to look up a per-processor port structure containing the list of processes that are ready to be used to run in the server's address space and to process the call (see Figure 5.3). One of the process descriptors is removed from the list and linked to the process descriptor of the caller to record the caller/callee relationship. Stack memory allocation and OUT page transfer are dealt with next (detailed in the next few sections), and control is then passed to the new server process which starts executing as the active process at its recorded starting address (i.e., its entry point).

When the server process completes, it issues a PPC return trap. No registers need to be saved this time, since the process is finished. The PPC trap handler uses the process descriptor of the returning process to locate the caller and the original port of the callee. The caller is unlinked from the callee, the callee is put back on the list of processes ready for a call on the target port, the callee's IN page is transfered back to the caller's OUT page if necessary, and any other resources, such as stack and OUT page memory, is returned. Finally, control is returned to the caller at the location just after where it made the original call.

The key benefits of this design all result from the fact that resources needed to handle a PPC are accessed exclusively by the local processor. By using only local resources during a call, remote memory accesses are eliminated. More importantly, since there is no sharing of data, cache coherence traffic is eliminated and no locking is required apart from disabling interrupts, which is a natural part of system traps. By eliminating memory, interconnection network, and lock contention, the PPC facility imposes no constraints on concurrency whatsoever.

We next proceed to examine in more detail the various steps outlined above.

5.3.2 Process Descriptor

The first step in handling a PPC call in the kernel is saving some of the current process's state in its process descriptor. As explained earlier, all PPC processing is done at exception level, which makes it more efficient, but introduces certain limitations and complications. In particular, running at exception level implies that locks cannot be acquired to protect the process descriptor from changes while the PPC is being processed. To deal with this limitation, process descriptors (PDs) have an additional component, called a process descriptor annex (PDAnnex) that can be accessed only by the processor the process is currently running on, and that is protected by disabling interrupts. Any operation on the PD object that requires access to the state in the PDAnnex must therefore be forwarded to the correct processor for handling. In addition, although the PD object itself is a clustered object and hence its implementation is fully hidden, the PDAnnex is exposed, allowing low-level code to directly manipulate it for saving and restore registers and accessing certain critical state.

Because PPCs are initiated by applications explicitly, the kernel need only save a couple of key registers (such as the instruction and stack pointers), so space for storing the full register set, in what we call a *savearea*, is not required. As a result, we only need as many full-sized saveareas (stored in a general pool) as there are processes currently in the ready queue.

5.3.3 PortAnnex table

Associated with each Port clustered object is a per-processor structure called the *PortAnnex* (see Figure 5.4). The PortAnnex in effect maintains a cache of processes to use for PPCs, and is built up as needed. It is analogous to the PDAnnex in that it holds per-processor information associated with the Port, can only be accessed on its respective processor, is protected by disabling interrupts, and has an open implementation to allow efficient low-level access. The PortAnnex contains a pointer to a list of PDAnnexes, representing the set of processes (also referred to as *workers* when they are discussed in the context of being the target of a PPC call) that are available for calls to the port. It also contains an "info" field that identifies the virtual processor associated with this list of workers, some other status bits, and a pointer to the next list of PDAnnexes for the case where multiple virtual processors of a program are scheduled onto the same processor (discussed in more detail in later sections dealing with remote PPCs).

To locate the PortAnnex given a PortID we chose the simplest and most time-efficient approach. Because the PortAnnex is so small (just a couple of pointers), we preallocate an array of PortAnnex objects to hold all possible ports. We also replicate this array to each processor so that each

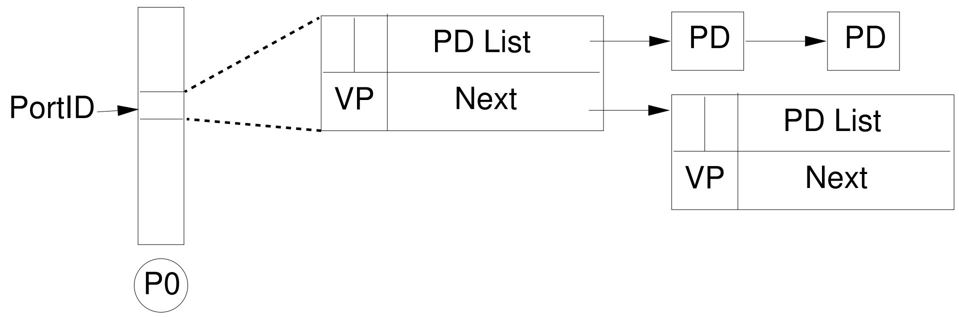


Figure 5.4: *This figure shows a closeup of one of the PortAnnex structures for a given processor ($\boldsymbol{P0}$). It includes the virtual processor number, some status bits, a pointer to the list of process descriptors (PDs) ready for a call, and a pointer to the next PortAnnex for this PortID for the case that multiple virtual processors from the same program have been scheduled on to the same processor.*

processor can have its own list of workers to handle a PPC call. With this design, a PortID is simply a direct index into the local PortAnnex table. Since authentication is performed by the server, there is no need to protect access to the ports. Clearly, in a very large system, a fixed sized table would be inappropriate, since it would need one entry for every running program and there could be thousands of programs running in a large system. One choice would be to replace the direct index with a hashed index. To avoid increasing the cost of a PPC call for the most critical servers, another alternative would be to have a small fixed sized table for the Ports of privileged servers and a hash table for the rest.

5.3.4 Stacks

After removing a worker from the PortAnnex worker list, a stack for the worker needs to be allocated. Stacks are managed in a special way in Tornado in order to maximize the performance of PPCs and provide the flexibility needed by different types of applications. Instead of each process having a stack region similar to the other memory regions of the address space, stacks are part of a special region of the address space that allows fast and efficient modifications to the virtual-to-physical mappings.

By default, each PPC worker on the PortAnnex list has a region of virtual memory set aside for its stack, but no physical or swap space backing it. When the process is selected to handle the call, a chunk of physical memory held in a special list for the PPC subsystem is mapped in as the stack of the process and, as an optimization, preloaded directly into the TLB. When the call completes, the physical memory of the stack is unmapped from the TLB and put back on the list.

This approach to managing the physical memory of PPC stacks has a number of benefits. Since the physical memory backing the stacks used by the worker processes is not bound to particular workers or even particular servers, but instead are assigned to workers on an as-needed basis, they are effectively recycled on each call. This improves the overall cache performance of the system, due to the smaller cache footprint that arises when multiple servers are called in succession and sequentially share physical stack pages. It also reduces the physical memory requirements of the system, since multiple servers called in succession may share a single stack, and extra stacks created during peak call activity can easily be reclaimed.

A concern of reusing stacks in this way is the possible security risk of sharing stacks amongst potentially untrusting servers since the stack memory is not automatically cleared between uses. This is currently addressed by permitting workers to permanently hold on to a stack (specified at Port creation time), allowing them to safely put sensitive information on their stack. A possible compromise solution would be to collect servers that trust each other into groups and only share stacks between servers in the same group.

A limitation of this approach is that stacks are not part of the regular shared address space of the program, and so variables on the stack can not be shared with other processes in the program. This has not been a problem in practice, however, since sharing memory on the stack is often the source of poor performance and bugs, and so sharing data on the stack is generally discouraged. Nevertheless, it is still something that must be considered when designing servers in Tornado.

Another limitation of our current implementation is that we restrict stacks to a small fixed size of 8KB on our platform. Although this has proven sufficient for all of our current servers, we have considered a number of ways of addressing this limitation. For example, it would be straightforward to support stack sizes of some fixed multiple of the page size, chosen on a service by service basis. It simply requires mapping as many pages as required into the appropriate location. For speed, this would be treated as an exceptional case.

Another approach would be to keep the current implementation and simply assign a larger virtual space for the stack. Accesses beyond the first page would result in a page fault and be handled by the normal page-fault handling mechanisms. This would keep the common case fast and only penalize those servers that require the extra space (which are likely to execute longer and more easily amortize the cost of the page-fault). Cleanup on return in this case, however, would require the extra pages to be returned to the system, though this can be implemented so as not to slow the common case.

Finally, it is also possible for the server to acquire a larger stack internally if required, using whatever stack management strategy is appropriate for the server, albeit with a corresponding increase in execution cost for each call. This is in fact the approach currently used for most regular (non-server oriented) applications that generally require larger stacks and whose main process survives for the duration of the program.

5.3.5 IN/OUT pages

After acquiring physical memory and mapping in the process's stack, the next step is to handle the IN/OUT pages. As with stacks, the IN/OUT pages are part of a special region of the virtual address space that allows efficient mapping and unmapping of physical memory. Normally, the OUT page of the caller is simply unmapped from its address space, which just involves clearing the appropriate TLB entry on the local processor and clearing a single word in the PDAnnex, and mapped into the IN page region of the callee address space in the same manner as the stack. On return the reverse happens. Because of the restrictions on the IN and OUT pages (as for the stack), the IN/OUT pages can only ever be accessed by the process they are mapped into, but this allows the bookkeeping to be kept simple and efficient.

5.3.6 Scheduling

Finally, once the memory of the stack and IN/OUT pages have been set up, the calling process is linked to the worker, and a context switch to the worker occurs, causing it to become the active process. In the case of an asynchronous call, the caller is put back on the ready queue rather than being linked to the worker. The context switch to the worker is actually nothing more than recording the identity of the currently running process and loading a couple of registers (the stack and instruction pointer of the worker; all other registers are either scratch registers or parameters from the calling process).

5.3.7 Exceptional conditions

So far, we have covered the common PPC call-handling cases, when all resources are available. The general principle in our design is to make this common-case path as efficient as possible, putting everything that is not part of the common case off to the side. For example, if, when allocating a stack for a worker, there is no memory left in the local pool, the call proceeds as normal and the lack of a stack is handled when the process page faults on the missing stack.

A few exceptional conditions must be handled more directly, however. The most common exceptional condition is an empty PortAnnex worker list, which can happen either because: ($\boldsymbol{i}$) this is the first access to the port on this processor, ($\boldsymbol{ii}$) the port has been accessed before but currently has no available workers, or ($\boldsymbol{iii}$) the port has been marked for remote PPC redirection. On the fast path, all three appear the same, but once detected, they are split into three separate cases. Case ($\boldsymbol{iii}$) is discussed further below. The first two are handled by redirecting the original PPC to a special Port, the *MetaPort*, whose sole purpose is to handle these types of special cases. The workers for this port have resources pre-reserved for them so that these redirected calls will always succeed.

When a call is redirected to the MetaPort, the MetaPort worker checks the reason for the redirection and calls an appropriate object to handle the situation. For example, in the case of an empty worker list, a call is made to the Port object which creates a new worker and adds it to the list. When the operation has been completed, the original caller is restarted from the point of the original call, effectively repeating the call. The second time around, the resources required should be available.

The key to this approach is that none of the complicated cases need to be handled at exception level during the actual PPC call; all the complicated work is redirected to special servers dedicated to the job.

5.3.8 Hardware exceptions

The PPC facility is not just used for inter-processor communication, but also for redirecting hardware exceptions (interrupts, page faults, process exceptions) to a port for handling. The design of the PPC facility is sufficiently general that exceptions of any kind can be directed to any port, whether to the kernel port (for page faults), some device server's port (for interrupts), or, in the case of process-generated software exceptions, to any program, such as the program issuing the exception or a debugger.

The implementation of exception upcalls is almost identical to the second half of an asynchronous call. In the simplest case, the exception is treated as an asynchronous call trap: the caller is put back on the ready queue, a worker is removed from the appropriate port, the parameters are filled in as requested earlier when the server registered to receive the exception, and the worker is made the active process and scheduled to run.

The key complications that arise in handling exceptions are due to priority issues and lack of resources. The priority issue comes about from the need to decide whether to first schedule the target of the exception (the worker on the port the exception is sent to) or the currently running process. We chose to have the exception handler run if it has the same or higher priority when compared to the process being interrupted, on the assumption that the handler may be more sensitive to latency than the process being interrupted (for example, if the interrupt is a notification of the arrival of a network packet or the completion of a disk request).

Dealing with lack of resources when making upcalls for exceptions is more difficult. There are two primary cases to consider: those where the exception is caused by the process currently running, such as a page fault or floating point exception, and those independent of the current process, primarily device interrupts. The first case is handled as if the process itself had issued a PPC call to the handler by turning the exception into a regular PPC call. For the second case, an attempt is made to redirect the original request to the MetaPort to allocate the needed resources as in regular out-of-resource cases. A timer event is also queued so that the interrupt handler can be retried in a few milliseconds. In the unlikely event that sufficient resources are unavailable even to take the above corrective action (which should be *very* rare) the best that can be done is to increment a count in a pre-allocated structure marking the fact that this interrupt has been missed. In general however, sufficient resources are pre-allocated to handle the expected interrupt rate.[4]

5.3.9 Remote PPCs

The final component of the implementation is remote PPC handling. For the common cases, as with local PPCs, the implementation of remote PPCs is quite straightforward. However, a number of complications must also be dealt with, many of them related to keeping consistent views of what is essentially distributed information. We begin by describing the common case scenarios, and then examine some of the more exceptional conditions.

Common case

As for the local case, we assume initially that all resources and information required to complete the call are available at the time of the call. A remote PPC starts off essentially as a regular PPC. When the PortID is used to look up an entry in the port table, it encounters one of two conditions that cause it to turn the PPC call into a remote PPC: ($\boldsymbol{i}$) the PortID provided by the caller directly specifies a particular virtual processor for the target that is not located on the current physical processor; or ($\boldsymbol{ii}$) the port itself specifies that the PPC should be redirected to a specified virtual processor. In either case, the result of the search is a target virtual processor for the port.

The virtual processor (VP) must then be mapped to the correct physical processor, since the mapping of virtual-to-physical processor is different for each application and can vary over time for any given application. The mapping is performed with the aid of a per-processor (set-associative) cache that maps $\langle$port, vp$\rangle$ pairs to physical processors. A miss in the cache is treated the same as many other exceptional cases, namely by redirecting the call to the MetaPort for handling. The MetaPort worker, in turn, asks the original target Port object for the mapping information which in turn requests the information from the program it is associated with. (The program is the final repository for this information.) The cache is then updated and the original request restarted.

Equipped with the identity of the target physical processor, the real work can begin. Transfer of control to a remote processor is accomplished using a low-level remote interrupt facility. The facility is based on inter-processor interrupts provided by the hardware. A simple interface is provided above the hardware that allows an arbitrary kernel function to be invoked on a given remote proces-

[4]This is essentially the same requirement as bounding the interrupt handling time to avoid missing interrupts in more traditional systems.

sor with a fixed number of parameters passed to it. The call is performed with interrupts disabled and the calling processor spins until a response is received from the remote processor, so remote interrupt calls must be short. To prevent deadlock (for example, in the case where the remote processor is sending the calling processor an interrupt at the same time) the calling processor periodically checks for (and processes) remote interrupts while waiting for its call to complete. When a remote interrupt is received, the call is processed at exception level without blocking (like all interrupts and exceptions).

The remote interrupt passes along the target PortID, the original PPC arguments, and, if required, the physical address of the OUT page and the amount of data in the OUT page to be copied. The remote end then handles it essentially as an interrupt upcall, but first sends back an acknowledgment to the calling processor before starting the new process so that the calling processor can continue operation. The remote end also marks the worker process as having been called from a remote node in order to be able to handle the return case properly.

On return, the flag indicating the process was called remotely is checked[5] and a remote interrupt is sent back to the calling processor (whose identity was recorded when the call was received) passing back the return arguments and the IN page address. The receiving end of the return interrupt then reschedules the blocked caller either by putting the caller on the ready queue, or by issuing an upcall directly to the caller, depending on the respective priorities of the caller and the currently running process.

Remote PPCs are thus like regular PPCs, but with a pair of remote interrupts on the call and return to connect the two sides. One key difference, however, is that a full context switch is required on both sides for both the call and return. One natural optimization we have not yet applied would be to have the caller spin in the trap handler for a few microseconds before calling the scheduler in case the remote PPC call completes quickly, avoiding the overhead of the two context switches on the calling side.

Less common cases

As with the local case, there are a number of less common cases that need to be handled specially. Most of them are handled in much the same way as in the local case, namely by redirecting the call to the MetaPort, which rectifies the problem and restarts the caller to retry the request. However there are some differences in the remote case. For example, if the remote port entry is empty, a reply is returned to the calling processor (during the remote interrupt processing) indicating that it will get a call back later when the port entry has been filled. An upcall is then made to the MetaPort on the

[5] Actually, there is a common flag for all exceptional conditions which is used to make the common case check fast, after which a more detailed check of specific flags is performed.

remote processor, which performs the normal allocation and then issues a remote restart call back to the originating processor. However, if the MetaPort worker, due to unavailability of resources, cannot handle the call, then a retry response is sent back to the caller, and the calling process is forced to delay for a few milliseconds before retrying the request. We expect this situation to occur rarely.

Another possible situation is that by the time the request arrives at the remote processor, the target program has migrated to another processor. In that case a response is sent back informing the caller to update its virtual-to-physical cache and retry the request. This is then handled as if there had been a cache miss at the beginning of the call.

Exceptional cases

Finally, there are some (hopefully very rare) situations that require more careful attention to get right. For example, when the caller or callee is terminated in the middle of a call, careful sequencing is required to ensure that deadlocks and inconsistencies do not arise.

Another exceptional case involves process migration. When a process migrates,[6] all outstanding local calls to or from the process must be converted into outstanding remote PPCs, and all remote PPCs must have their status updated to indicate the new location of the process. Note, however, that this feature has been designed but not yet been implemented.

5.4 Performance

Table 5.1 presents performance results for various PPC operations in terms of instructions and measured cycles on NUMAchine. A subset of these operations are broken down into the instruction counts of their key components in Table 5.2. The breakdown was obtained by collecting SimOS traces and manually inspecting them. Each of the operations in Table 5.1 is discussed in detail in the following paragraphs.

The first entry in Table 5.1 is for a regular PPC call, in which all resources to complete the call are available at the time of the call. Although the design is primarily targeted at maximizing concurrency, it still compares favourable to some of the fastest uniprocessor IPC times. For example, two one-way IPC calls for the L4 (uniprocessor) IPC (considered one of the fastest IPC implementations in existence today) running on the same (MIPS R4000 based) processor require 158 instructions [Liedtke *et al.*, 1997]. The reported times for L4 include only the time spent in the kernel handling the IPC requests. If we look at the column for *regular* PPC calls in Table 5.2 and sum up

[6] Actually, it is the process' virtual processor that migrates, taking all its processes with it.

Operation	Instructions	Cycles
PPC call	377	695
PPC call IN/OUT	459	943
Empty port call	3830	19522
First PPC call	4116	33756
Interrupt latency	814	1618
Remote PPC call	1832	7016

Table 5.1: *This table compares the number of instructions required to perform a few types of PPC calls as measured under SimOS to the number of cycles as measured under NUMAchine.*

	regular	in/out	remote
stub overhead	14	14	16
save/restore	50	50	50
ppc call trap handling	50	50	50
debugging	30	30	30
stack management	62	62	91
IN/OUT transfer	—	82	—
raw PPC overhead	117	117	721
xobject checking	49	49	93
remote interrupt	—	—	781
Total (instructions)	377	459	1832

Table 5.2: *Instruction count breakdown of some more common PPC operations.*

all the kernel components involved in a call, we have 259 instructions. If we further discount the debugging code (30 instructions provided to ease development and debugging) and the stack management code (62 instructions, which are optional if stack sharing is disabled for a server), we have a count of 167 instructions, reasonably close to the 158 instructions of L4. In addition, with PPCs, 8KB of data can be exchanged in both directions (IN/OUT PPC calls) at an extra cost of only 82 instructions.

From Table 5.1 we also see the cost of calls to an empty port (19522 cycles) and to a non-initialized port (33756 cycles). These costs are quite high, but they have not been optimized in any way. Calls to an empty port involve extra overhead, primarily for creating a new worker and restarting the call, while calls to an uninitialized port involve one-time overheads beyond that, for example to initialize the virtual processor data structures. In microseconds these measured times amount to 133 microseconds for an empty port call, and 227 microseconds for a call to a non-intialized port, which compare reasonably well to the time of 370 microseconds required for creating a kernel thread under SGI IRIX 5.3 on a system with the same processor and a faster memory subsystem.

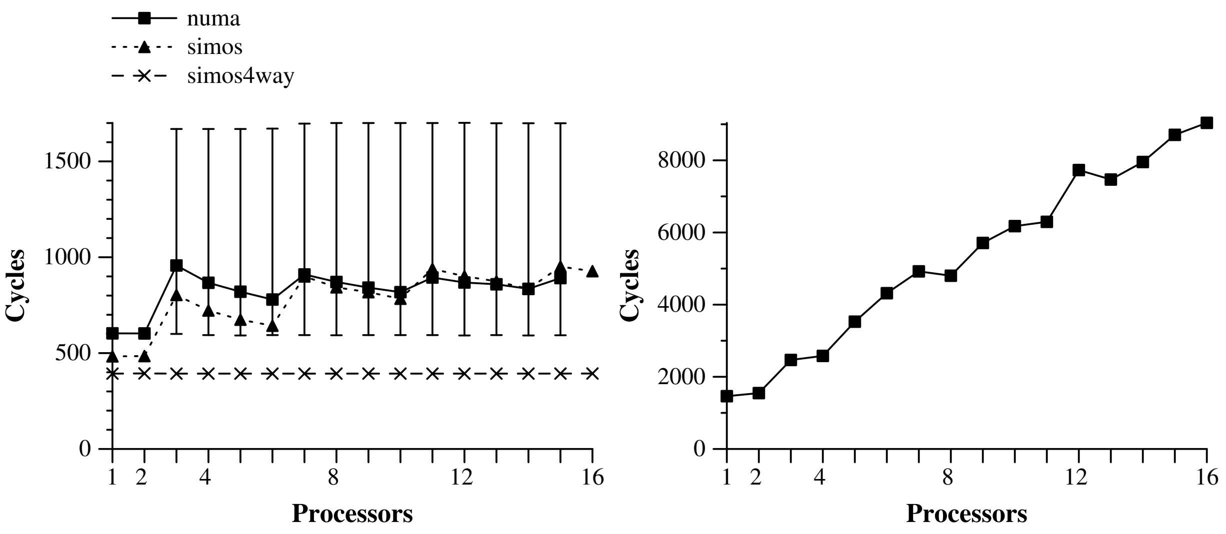
The entry *Interrupt latency* in Table 5.1 shows the cost for interrupts delivered by way of a PPC to a user-level server. The cost of 1618 cycles (or almost 11 microseconds) is somewhat high. However, approximately 30 percent of the cost is due to the Tornado interrupt infrastructure that was designed for flexibility and ease of development (rather than speed), and is independent of the use of PPCs as the final interrupt delivery mechanism. Also, the core PPC components used in the case of an interrupt have not been optimized, and hence are several times slower than they should be. It should be possible to reduce the cost to that of a regular PPC call (likely even less), allowing interrupts to be efficiently redirected to any server, not just the kernel.

The remote PPC case (shown in Table 5.1 and broken down in Table 5.2) is somewhat more complex than the others, as it involves two processors, remote interrupts, full context switches, and a certain amount of overlap.[7] The total cost of 7016 cycles includes approximately 1500 cycles just for communicating parameters and status information between the two processors, which entails four cache misses, a pair for each of the call and return. A large part (more than a third) of the rest of the cost is in the general remote interrupt facility. The raw PPC component is also large, due to the use of an un-optimized version for the remote case (the same one used for PPC interrupts), and can be expected to be reduced to roughly the same cost as a local PPC (117 instructions vs the current 721). The same holds for the xobject checking component, which should be faster for the same-program remote case than for the cross-program case, but this has not yet been optimized.

Despite these costs, remote PPCs still compare favourably with other highly optimized remote communication facilities, such as that provided in the Hive operating system [Chapin *et al.*, 1995b]. They report best-case times of 34 microseconds compared to our time of 47 microseconds. However, their system includes a faster processor (200MHz vs. 150MHz) with set-associative caches, a faster memory system (700ns memory access time compared to our 1800ns time), and a faster interprocessor interconnect (1.2 GB/s vs. 320 MB/s). Their system also includes special hardware support for communicating small messages between processors (requiring only 1 microsecond to send a cacheline vs. approximately 2.5 microseconds in our case). It is also a specialized facility available just for the kernel, rather than a general purpose facility available to all programs.

As stated above, PPCs were primarily developed for concurrency. Figure 5.5(a) shows the performance of concurrent PPC calls to a single port from 1 to 16 processors on three different platforms. The range bars for the NUMAchine case (*numa*) indicate the range of times taken by different processes, with the markers indicating the average across all processes. Although the average performance is reasonable, it is not completely flat and there is a wide variability in process times. The times for SimOS (*simos*) show a similar pattern, again providing confidence in the simulator at reflecting the various quirks of the system. As was the case with the clustered object system, cache

[7] Note that the instruction counts reported include only those components that are on the critical path of the call, and does not include some of the cleanup code that takes place concurrently with critical path work on the other processor.



a) Concurrent PPCs a) Concurrent PPCs with shared write

Figure 5.5: *The figure on the left shows the performance for a test of concurrent PPC requests to a common server on NUMAchine (*numa*), SimOS (*simos*) and SimOS with 4-way set-associative caches (*simos4way*). The figure on the right shows the same test, this time only on NUMAchine, with a single shared write inserted in the critical path.*

conflicts are the primary reason for this behaviour, as the results on SimOS configured with 4-way set-associative caches (*simos4way*) demonstrate: SimOS with 4-way set-associative caches shows almost perfect scalability.[8]

To give an impression of the importance of the various optimizations for improved concurrency that have been implemented within the PPC subsystem, Figure 5.5(b) shows the performance of the same concurrent PPC tests on NUMAchine, but this time with the PPC call code modified to include one extra write to a shared word. In this case, the time for a call increases linearly with the number of processors. This illustrates the importance of eliminating accesses to shared variables for maximum performance and concurrency, as performance suffers significantly, even with just a small number of processors.[9]

[8]The stair-step pattern in the results is due to some processors having a large number of cache conflicts and others almost none at all. As a processor with cache conflicts is added to the test it brings the average up, while when a processor with no cache conflicts is added it brings the average down. The reason certain processors (for example, processors 3, 7, 11, and 15 in this test) experience cache conflicts while others don't is unclear, but is likely due to the pattern of memory allocation in the test.

[9]Note that no sychronization is used for accesses to this shared variable, making the example artificially optimistic.

5.5 Open Issues

As with the clustered object system, there are a number of open issues in the PPC system (most of which have been alluded to earlier):

- The use of a shared stack pool provides a number of potential benefits in terms of improved cache hit rate (through a smaller cache footprint) and reduced memory requirements, but the security issues remain to be addressed. In addition, the actual benefits need to be evaluated in a realistic setting to determine their true value.

- The other limitation of the shared-stack approach, is the limit placed on the size of the stack. Although a number of options were proposed for elimintating this restriction, they have yet to be fully designed or evaluated. However, it is has so far not posed a problem with any of our servers.

- Finally, multicast PPCs would be very useful for the propagation of changes to distributed and replicated clustered objects.

5.6 Related work

The majority of research on performance conscious interprocess communication (IPC) has been for uniprocessor systems. Excellent results have been reported for these systems, to the point where it has been argued that the IPC overhead has become largely irrelevant [Bershad, 1992].[10] Although many results have been reported over the years on a number of different platforms, the core cost for a call-return pair (with similar functionality) is usually between 100 and 200 instructions [Liedtke, 1993, Ford and Lepreau, 1994, Hamilton and Kougiouris, 1993, Engler *et al.*, 1995]. These implementations, and ours, apply a common set of techniques to achieve good performance: $(\boldsymbol{i})$ registers are used to directly pass data across address spaces, circumventing the need to use slow memory [Cheriton, 1984]; $(\boldsymbol{ii})$ the generalities of the scheduling subsystem are avoided with hand-off scheduling techniques [Black, 1990, Cheriton, 1984]; $(\boldsymbol{iii})$ code and data is organized to minimize the number of cache misses and TLB faults; and $(\boldsymbol{iv})$ architectural and machine-specific features are exploited or avoided depending on whether they help or hinder performance.

The PPC facility goes beyond the application of these techniques, optimizing for the multiprocessor case by eliminating locks and shared data accesses, and by providing concurrency to the servers. In a multiprocessor, accesses to shared data can result in cache misses or increased cache

[10] We do not agree with these arguments.

invalidation traffic which can add hundreds of cycles to the cost of an operation. (The relative cost of cache misses and invalidations is still increasing as processor cycle times are further reduced.) With the increases in efficiency of IPC implementations, locks can quickly saturate, even if the critical sections are very short. Multiprocessor microkernel operating systems depend heavily on maintaining concurrency in the servers as well as in the kernel and hence require a highly concurrent IPC facility that supports concurrency in the servers. Tornado addresses all of these key multiprocessor issues while none of the existing systems address any of them.

The key previous work done in multiprocessor IPC was by Bershad et. al. [Bershad *et al.*, 1990], where excellent results were obtained on the hardware of the time. However, it is interesting that the recent changes in technology lead to design tradeoffs far different from what they used to be. The Firefly multiprocessor [Thacker and Stewart, 1987] on which Bershad's IPC work was developed has a smaller ratio of processor to memory speed, has caches that are no faster than main memory (used to reduce bus traffic), and uses an updating cache consistency protocol. For these reasons, Bershad found that he could improve performance by idling server processes on idle processors (if they were available), and having the calling process migrate to that processor to execute the remote procedure. This approach would be prohibitively expensive in today's systems with the high cost of cache misses and invalidations.

5.7 Summary

The new IPC facility we developed for Tornado is based on the Protected Procedure Call (PPC) model rather than a message passing model. In the PPC model, a client process is thought of as crossing directly into the server's address space when making a call. This model together with our implementation has a number of important properties. First, the model inherently provides as much concurrency in the server as the requesting clients, while the implementation uses no locks in the common case thereby imposing no constraints of its own on concurrency. Second, no shared data is accessed in the common case, minimizing cache consistency traffic. Finally, the model dictates that requests are always handled on the same processor as the client, allowing the server to keep state associated with the client's requests local to the client. With our implementation, the resources provided to the server to handle a request (in particular, the server's stack) are local to the processor on which the request is being serviced, hence minimizing implicit accesses by the server to shared data.

Chapter 6

Support Infrastructure

A multiprocessor operating system (including its system servers) depends on a significant infrastructure which can strongly influence its performance. Two key components are the locking facility and the memory allocation facility (malloc/free in the C language or new/delete in C++). Particularly for an object-oriented microkernel multiprocessor operating system, such as Tornado, these two facilities can severely limit performance if not implemented carefully.

A multiprocessor operating system has special requirements compared to most programs:

- It has highly concurrent demands and therefore must be carefully structured to avoid bottlenecks.
- It spans the entire system and therefore must be scalable and efficient in its use of resources.
- It must efficiently support a highly diverse set of workloads.
- It runs continuously and therefore must manage resources carefully to avoid leaks and imbalances in their distribution and availability.
- It must be highly robust, both in terms of reliability and performance.

The supporting infrastructure must take these various requirements into account.

In this chapter, we look at the issues surrounding the design of the locking facility and the memory allocation facility for Tornado, and describe and evaluate the design choices made for each.

6.1 Locking facility

6.1.1 Locking issues

Although there are a large number of issues to consider when designing a locking facility, we were primarily concerned with the following four:

- Locking overhead: we wanted to minimize both the time and space overhead of locking.
- Lock scheduling: we wanted to provide certain guarantees (or probabilistic assurances) about the order in which contended locks are acquired by processes waiting for the lock.
- Deadlock and races: it is critical to provide a structured environment for avoiding deadlock and race conditions.
- Miscellaneous: we needed to deal with exception level interactions, supporting both user and kernel services equally, and consider the potential benefits of a lock-free approach to concurrency control.

We examine each of these issues in detail.

Locking overhead

One of the primary concerns of a locking subsystem is to minimize the time and space overhead of locking; locking does not provide any end-user functionality and hence is all strictly overhead. Although processor cycles spent waiting for a held lock can be justified in that no forward progress can be made anyway, time spent acquiring an uncontended lock is pure overhead and should be kept to a minimum (particularly assuming that the system is designed to minimize contention in the common case). Our approach is one that is commonly used [Karlin *et al.*, 1991], namely to use two different algorithms on the same lock (also known as a two-phase or adaptive lock): a fast algorithm built on a simple test-and-set style atomic sequence with spin-based exponential backoff, and a slower but more scalable algorithm that places waiting processes in a queue. The key is to combine these two algorithms without adding time or space overhead to the common (fast) case.

Although we are mostly concerned with minimizing the cost of the uncontended case (which we expect to be the common case in Tornado), we are still concerned with reducing the cost of contended locks. Of particular concern is minimizing the secondary effects that spinning processes can have on performance, on both the lock holder and the rest of the system. This can occur, for example, as a result of memory accesses that the waiting processes issue while spinning, causing excessive interconnection traffic and slowing the entire system. In particular, if the lock and the data are co-located on the same cache line (in order to bring the data to the lock holder on lock acquisitions), false sharing occurs between the lock holder accessing the data and the waiting processes accessing the lock, significantly increasing the lock hold time and forcing the waiting processes to wait even longer. This argues for a design that ensures that waiting processes do not repeatedly access (even for read-only purposes) the lock or any other shared variable. Although we currently use a simple spin-then-block approach, we consider some alternatives in Section 6.1.4.

By comparison to reducing the time overhead, reducing the space overhead of locks is comparatively easy. It is even possible to reduce the space overhead to zero for simple atomic operations, such as add, bitset, or enqueue, using primitives such as compare-and-swap or load-linked/store-conditional. When a lock is needed, the overhead can be reduced to one or two bits, allowing the combination of data and its lock in a single word, provided there is a spare bit or two available[1] and provided contention is low enough that false sharing is not a concern. This will be described in detail in the next few sections. Although the space overhead may not be a major factor if the lock granularity is fairly coarse, the lower the space requirements, the more flexibility the locking system affords the designer.

Lock scheduling

Another important issue is the type of scheduling to provided to the processes waiting for a lock. Generally the options fall into three broad categories: i) random; ii) first-in-first-out; and iii) highest priority first. The first is typical of traditional spin-lock implementations, where the next process to acquire the lock is essentially random (depending on the spinning backoff algorithm). The second is usually the by-product of some sort of queue-based implementation, where the lock holder explicitly hands-off the lock to the process at the head of the queue. The final one is a variation on the queue-based implementation, where the order in the queue is determined by something other than arrival time.

The most important scheduling issue (from our point of view) is starvation and overall fairness (ensuring each processor has an equal probability of acquiring the lock). It is important that no thread be denied a lock indefinitely, and that overall, the waiting time for a lock is evenly distributed over all acquirers. Queue-based locks are generally the best to guarantee this, although properly tuned spin-locks are statistically just as good with lower uncontended overhead. Again, adaptive schemes should be able to achieve both.

[1] For example, pointers often must be aligned to word boundaries, leaving the bottom few bits available for lock bits.

Deadlock and races

Although not part of the actual locking implementation, the overall locking strategy used and the choice for concurrency control are tightly intertwined with the methods required for dealing with deadlock and race conditions. Typically, rigidly defined locking hierarchies are needed, where specific classes of locks must be acquired in specific orders. As a basic rule of thumb, the shallower the lock hierarchy, the more flexibility there is in defining the system structure. Moreover, being able to rely more on leaf-locks (locks under which no other locks are ever held) eases many of the locking protocol decisions. Unfortunately, many systems with deep lock hierarchies restrict the ability to use leaf-locks (or lock-free approaches, as both are generally used under the same conditions).

The Tornado clustered object system significantly simplifies the locking protocol by allowing locks to be dropped and acquired under many more conditions than a typical system. This is because there is no danger of objects being destroyed if their lock is dropped, making it safe to later access the object even if it is being destroyed. Of course, the object state needs to be rechecked, since it may have changed between the time the lock was dropped and reacquired. As a result, locking in Tornado generally only needs to consider the issues of concurrent access to a data structure within an object, and not the relationship between locking decisions in different objects.

Miscellaneous

Three separate but related issues that need to be considered are ($\boldsymbol{i}$) the interaction between the locking protocol and exception handlers, ($\boldsymbol{ii}$) the use of a lock-free approach to simplify certain locking issues, and ($\boldsymbol{iii}$) the need for a common framework for the kernel and user-level servers, including debugging and performance monitoring.

Because an interrupt handler can run at arbitrary times, synchronization between regular kernel processes and the interrupt handler are required. The normal way of dealing with this is to disable interrupts around data structures that are shared between the two levels. Another approach is to use a lock-free approach, which doesn't suffer from the possibility of deadlock [Massalin and Pu, 1991]. Unfortunately, as we shall see, the limitations of current hardware prevent us from using lock-free concurrency control as a general solution to the problem, even though it would offer many benefits.

The final issue—providing a common framework for the kernel and user-level—is important because system servers face all the same issues as the kernel. They both need the ability to monitor the performance of the locks and determine who is holding which locks when the system fails.

6.1.2 Tornado locking facility

The design of the Tornado locking facility is heavily influenced by the clustered object architecture. First, clustered objects encourage localization and distribution of objects, reducing the amount of concurrency any single object (i.e., clustered object representative) must support. Second, the encapsulation encouraged by the clustered object framework requires each object to do its own locking, requiring fairly fine grained locks and necessitating a time and space efficient implementation. Third, the clustered object garbage collection system removes the need for much of the complex locking protocols, since objects can hide their locking protocol entirely within their own structure and locks can be safely dropped under most circumstances.

These factors have led us to focus on space and time efficiency and less on the scalability of any one lock. The encapsulation of locking decisions within each object means that most locks are leaf-locks, allowing additional optimizations (such as the replacement of locks with lock-free approaches) and limiting the time most locks are held. Hence we favour spin-then-block bit-based locks for most purposes, and lock-free approaches where convenient.

The next few sections discuss the particular implementation issues the Tornado locking facility faced.

Lock-free support

Tornado's lock free support is built on the load-linked and store-conditional instructions provided by the MIPS processor used in NUMAchine.[2] The semantics of the two instructions are as follows: the load-linked acts as a regular load instruction but in addition sets a marker (in the processor) indicating the address of the memory location loaded. If, at any time prior to the store-conditional, the processor detects a store to the same location by another processor (for example, through an invalidation for that cache line) or if an exception of some sort occurs (such as a timer interrupt), the marker is cleared.[3] When the program issues a store-conditional, the store only proceeds and is successful if the marker is still set. If unsuccessful, this is relayed to the program so it can retry the sequence. The equivalent of spinning on a lock thus occurs through the retry mechanism.

The two load-linked/store-conditional instructions allow a large number of atomic primitives to be synthesized. For example, a classic test-and-set instruction is implemented by doing a load-linked, testing the value loaded, and if it is clear, doing a store-conditional of one, repeating the sequence if the store conditional fails. Although there are restrictions as to what sequence of instructions can come between a load-linked and a store-conditional, most of the basic primitives,

[2] Most modern processors provide equivalent instructions.

[3] The marker is overwritten if another load-linked instruction is executed prior to the store-conditional. Thus load-linked/store-conditional pairs cannot be nested.

such as swap, compare-and-swap, fetch-and-set-bit, fetch-and-add, can be implemented in a similar way.

Because the two instructions are for the most part just load and store instructions, they don't slow down the processor and if the target data is in the cache, the instructions execute at processor speeds, rather than at bus or memory speeds.

There are many other benefits to using these primitives. For example, in the case of simple atomic operations like fetch-and-add, they can be used to eliminate the need for a lock altogether, by embedding the critical section code between a load-linked for the data and the corresponding store-conditional. The elimination of the lock not only saves space, but is more efficient, since it eliminates much of the locking overhead. The other advantage is that deadlocking with an interrupt handler cannot occur since the load-linked marker is cleared on an interrupt (making the sequence an optimistic transaction).

In Tornado we use this *lock-free* approach for many of the simple cases like incrementing shared counters or atomically updating certain bits in a word. We also use it in more specialized situations, such as for optimizing the locking operations of the memory allocator free lists (described in detail later in this chapter). In this case, we used the load-linked/store-conditional instructions to construct a special fetch-and-lock routine that both locks the free list and returns the head of the list, but only if the list is not empty and not already locked. This allows the common case, where the list is not locked nor empty, to be implemented very efficiently, relying on a more general purpose routine to handle all the exceptional conditions.

One of the drawbacks to the lock-free approach is that few options exist for dealing with high contention besides retrying the entire operation until it succeeds. Another drawback is that since the sequence of instructions that can be inserted between a load-linked and a store-conditional instruction is limited, only a limited number of atomic operations can be synthesized (these restrictions are common to most load-linked/store-conditional implementations).[4]

Bit-based spin-then-block locks

The Tornado bit-based spin-then-block locks are two-phase locks that use only two bits in any word, leaving the other bits of the word untouched. Other lock variations, like spin-only and non-bit-based locks are also provided. These locks exploit the flexibility of the load-linked/store-conditional instructions for constructing atomic primitives. The implementation can be broken down into a few

[4]There are more complex algorithms that can bypass these problems [Herlihy, 1993, Valois, 1995b], but they have other limitations.

simple steps.[5] A lock acquire proceeds as follows:

1. an assembly language routine does an atomic test-and-set on the first of two bits reserved in the lock word for the exclusive use of the locking routine (the rest of the bits may hold anything the user desires, such as a pointer or counter, and are left untouched by the locking routines);[6]
2. If the test-and-set is successful, then the lock has been acquired and we are done;
3. otherwise, we increment a spin count and if the second bit is not set and the spin count has not been exceeded, we go back to step 1;
4. if the spin count exceeds a limit[7] or the second bit is set, we give up spinning and enqueue ourselves (our process id) on a list of processes waiting for the lock, atomically set the second bit in the lock, and go to sleep. Because the lock consists of just two bits, the list is maintained in a separate structure, namely a hash table keyed by the lock address and the lock bit numbers;
5. when woken up (presumably by the previous lock holder) we verify that we are marked to get the lock next (going back to sleep otherwise), unlink ourselves, and return.

Releasing the lock is somewhat simpler:

1. atomically clear the first bit if the second bit is clear;
2. if the above succeeds, we have released the lock and are done;
3. otherwise, there may be someone queued up waiting for the lock who must be woken up;
4. if no one is queued, we clear both bits;
5. if someone is queued, we set a flag in the record of the next process in line for the lock, wake it up, and return.

[5] We leave out some of the precautions that must be taken to prevent race conditions within the lock implementation itself.

[6] The particular bits reserved are chosen by the user by instantiating the appropriate C++ template lock class provided by the locking facility.

[7] This limit is usually chosen so that the spin time is equal to the blocking time, ensuring that the overhead is never more than twice the optimal overhead [Karlin *et al.*, 1991]. Finding a spin count that corresponds to the desired spin time can be difficult however. We return to this issue in Section 6.1.4.

This design provides both a time and space efficient implementation for the common case of low-contention, with only a modest increase in time and space when there is contention.

Unfortunately, there was a subtle problem with the design that went unnoticed for a long time. The problem is related to the fact that although our locking facility ensured there were no races in the manipulation of the lock word within the locking system itself, it didn't consider races with other routines that might be manipulating the user data portion of the same word (recall that the lock subsystem is only given jurisdiction over two of the bits in the word). Under certain circumstances this could result in the second bit being accidentally cleared after having been set by a waiting process, thus leaving the waiting process in limbo after the lock is released, potentially indefinitely.

The solution to the problem requires that all other manipulations of the word use a specific protocol. Because the lock is often used to protect the rest of the contents of the lock word itself, we return the contents of the lock word (with the lock bits masked off) on a lock acquire, and allow the user to provide the updated value of the word on lock release. This ensures that all manipulations of the lock word are under the control of the locking facility and are properly synchronized.[8]

NUMA effects

For the most part, NUMA effects are not a major concern in the locking subsystem, since locks are normally contained within the objects they are meant to lock and the clustered object system already provides an environment for enhancing locality between the objects and the processes accessing them. The only place where it is an issue is with the hash table used for the bit-based blocking locks described above. For this case, we use a clustered-object-based hash table that is distributed across the set of processors with the address of the lock used to determine the appropriate hash table to use. Since the memory allocation facility (described below) partitions the address space to gain locality, the memory from which the lock was allocated can be determined and hence the hash table representative closest to the lock can be identified.

Lock debugging

Another important part of the locking facility is the debugging support that it provides. Two implementations of each type of lock are provided: a debugging version and a non-debugging version. The version chosen is controlled by standard compile-time debugging flags and can be overridden by explicitly choosing the type of lock-debugging desired on a lock-by-lock basis. The debugging locks keep track of information such as which processes are currently holding which locks, who is waiting for which locks, and the location in the program where each process acquired or is waiting

[8] Other solutions are clearly possible and may be preferable under some circumstances.

Operation	Instructions	Cycles	R10000
Uncontended simplest lock	10	12	10
Uncontended spin-only bit	20	31	10
Uncontended spin-block bit	25	36	11
Contended block-only word	3014	14236	—
Contended block-only bit	3100	17346	—

Table 6.1: *Time in instructions and cycles for NUMAchine, and cycles for an SGI MIPS R10000 machine, for a lock/unlock pair for a minimal (*simplest*) uncontended lock, for a bit-based spin-only uncontended lock, for a bit-based spin-then-block uncontended lock, for a two-processor contended full-word block-only lock, and for a two-processor contended, bit-based block-only lock.*

for the lock. This type of information is invaluable for detecting deadlocks and tracing lock dependencies. The infrastructure is also designed to support the collection of various statistics concerning contention and lock-holding time. No changes to the client code are necessary to switch between the two lock types except a recompile with the appropriate debugging flags.

6.1.3 Performance evaluation

Table 6.1 shows the number of instructions and NUMAchine cycles required for several basic locking operations, as well as the cycles required on an SGI Origin 200 using a more recent version of the same processor, the MIPS R10000. As was suggested earlier, a number of different approaches can be used for dealing with high contention. We primarily use spin-then-block locks with exponential backoff, but queue-based spin-only locks and other options are also available. Because it is relatively straightforward to adapt our two-phase locks to use any of the other alternatives for the second phase, we focus our attention in this section on the uncontended performance of the locks and the cost of the two-phase and bit-based aspects of those locks.

The *uncontended* tests in Table 6.1 measure the time to lock and unlock a free lock by a single process. The first entry in the table, *simplest*, is for a simple test-and-set lock with no backoff. The *spin-only bit* lock is a bit-based version of the *simplest* lock, and is used in Tornado in situations where the process cannot block. The *spin-block bit* lock is a bit-based two-phase lock. If we compare the two bit-based locks, we see that the two-phase lock pays a 5 cycle penalty even though there is no contention in these tests. This is due to the need to test for blocked processes when the lock is released. However, even the faster *spin-only bit* lock is over two-and-a-half times slower than the *simplest* lock (31 cycles vs. 12 cycles). This would seem to indicate that the bit-based locks should only be used where space is the primary concern. However, if we consider the same tests run on the R10000 processor, which is a super-scalar out-of-order version of the same processor used on NUMAchine, the differences all but disappear. This is because most of the extra work that these locks perform can be done in parallel with the basic function of the lock, and hence are not in the critical path. We conclude that the cost for a more flexible and space-efficient lock is minimal on the current generation processors, and not unreasonable even on the older processors of NUMAchine.

The *contended* results show the cost for two processes running on two different processors to handoff to one another through a blocking lock. In this test, one process tries to acquire the lock already held by the second process and hence blocks. The second process delays for a bit and then releases the lock, giving ownership to the first process and waking it up. The second process then immediately tries to re-acquire the lock. Since the lock is now held by the first process, the second process blocks, and the cycle repeats. The time to acquire and the time to release the lock are measured separately and added together to give the total time reported in the table.

The two versions shown are a *bit*- and *word*-based implementation of a blocking lock (i.e., the process blocks immediately if the lock is busy). This is necessary since we do not want to include the spin time in our measurements of the overhead of the lock. The bit-based version is the one previously described, in which only two bits in a word are used to indicate the state of the lock, and a queue of waiting processes is maintained separately in a hash table. The full-word version embeds the list directly in the lock, removing the hash table overheads.

Compared to the uncontended case, the cost of these blocking locks is high, as it includes blocking, remote wakeup, shared queue manipulations, and, in the case of the bit-based lock, shared hash table operations. The cost of the hash table operations adds little to the overall cost, since it is only paid if there is contention. Hence, with the addition of an appropriate spinning phase, the two-phase bit-based spin-then-block locks are reasonably space and time efficient.

6.1.4 Open Issues

The primary open issue still to be addressed in the Tornado locking facility is whether the spin-then-block locks provide adequate scalability as the system is scaled to a larger numbers of processors. Although Tornado is designed to avoid high-contention, it is still important that it gracefully handle contention when it does arise. However, with the framework described, it is relatively straightforward to use other adaptive locking techniques [Lim and Agarwal, 1994] without changing the performance of the underlying non-contended case. A three-phase lock is one approach, in which processes first spin on the lock bit, then enqueue themselves and spin on a flag in their queued structure, and then finally go to sleep. The middle phase avoids the cost of the sleep and wakeup calls as well as the negative memory effects of spinning on a central lock word. Another possibility to deal with cache-line traffic caused by excessive spinning is to use hardware event counters commonly available in current microprocessors to detect when spinning is causing cache misses, and reduce the rate at which the state of the lock is checked under those circumstances. Alternatively, cycle counters could be used to ensure that the spin phase is limited to a certain amount of time, rather than a certain number of loop iterations.

6.1.5 Related Work

Two of the most important issues with locking are correctness (avoiding race conditions and deadlock) and performance (reducing the cost of waiting for and acquiring a lock). Recent work on correctness has dealt primarily with either detecting races [Savage *et al.*, 1997] or avoiding deadlock [Paciorek *et al.*, 1991]. In the area of performance, there have been efforts at determining the best spin time for two-phase locks [Anderson, 1990, Karlin *et al.*, 1991], making locks adapt to contention [Lim and Agarwal, 1994], and dealing with multiprogramming issues [Kontothanassis *et al.*, 1997]. There have also been numerous proposals to replace the shared spin lock with a queued spin lock [Mellor-Crummey and Scott, 1991, Magnussen *et al.*, 1994].

There have been many papers written concerning experiences with locking in multiprocessor operating system [Campbell *et al.*, 1991b, Balan, 1992, Campbell *et al.*, 1991a, Saxena *et al.*, 1993, Peacock *et al.*, 1992, Presotto, 1990, Pike *et al.*, 1991, Ruane, 1990, LoVerso *et al.*, 1991], primarily detailing issues with retrofitting locking into existing Unix-like system.

Two interesting papers detailing experiences with novel operating systems discuss the use of an exclusively lock-free approach to operating system concurrency control [Massalin and Pu, 1991, Greenwald and Cheriton, 1996]. However, both used older processors that support hardware features not found in existing systems, and fail to provide a general framework in which arbitrary concurrency control requirements could be re-cast in a lock-free manner. Other work that has attempted to extend lock-free techniques to more general data structures has generally been either too expensive or required additional hardware support [Herlihy, 1993, Valois, 1995a]

One other locking paper that is of interest is our own previous experiences with locking in Hurricane [Unrau *et al.*, 1994] on the Hector multiprocessor [Vranesic *et al.*, 1991]. That paper argued for a hybrid approach to locking with coarse-grained locks protecting bits used for fine-grained locks. However, with changes both in the system software design between Tornado (fine-grained object-oriented concurrency control) and Hurricane (coarse-grained non-object-oriented structure), and in the underlying hardware between Hector (slow, in-memory, fixed atomic instructions) and NUMAchine (in-cache, flexible load-linked/store-conditional instructions) those previous techniques are no longer applicable.

The key to Tornado's locking strategy is the synergy between the different components, such as (i) the object-oriented design which contains contention to within individual objects, (ii) the clustered object system which further contains contention to within individual reps, (iii) the garbage collection system, which eliminates many of the locks and simplifies the concurrency control protocol by removing many of the potential race conditions, and (iv) the low time and space overhead locks along with the judicious use of lock-free techniques that match the fine-grain locking structure of the clustered object design.

6.2 Memory allocation subsystem

Memory allocation has a long history of study (well surveyed by Wilson [Wilson *et al.*, 1995]), but few studies have considered the particular issues faced by operating system software [Bonwick, 1994, McKusick and Karels, 1988, Sciver, 1990], and fewer still have considered the additional requirements of multiprocessor system software [McKenney and Slingwine, 1993]. In this section we examine the specific issues faced by the Tornado memory allocation facility and the design that arose to address these issues.

6.2.1 Multiprocessor allocator issues

Many issues must be considered when designing a memory allocation facility, both in terms of the internal design of the allocator and how it will be used. Some of the factors to consider include:

- Multiprocessor locality: limit sharing to those cases where it is absolutely necessary.
- False sharing: avoid placing data with different sharing patterns in the same cache line or page.
- Cache reuse: maximize reuse of data that is likely to be already in the cache.
- Concurrency control: maximize concurrency so as to (at least) match the scale of system.
- Exception level support: account for the need to allocate and free memory at exception level.
- NUMA awareness: maximize locality with respect to memory module access times.
- Internal/external fragmentation: minimize memory wastage due to fragmentation.
- Execution time: minimize time to perform allocations/deallocations.
- Return unused memory: maximize ability to return memory for general use.

We examine each of these issues in detail.

Multiprocessor locality

The key to multiprocessor locality is to use per-processor data structures wherever possible. In the case of memory allocation, using per-processor lists of free blocks improves multiprocessor locality both in the free-block management structures and in the memory being allocated itself. The latter point is particularly important with current systems' large secondary caches (up to 8 MB on some systems), since it increases the chance that memory that was recently accessed and freed on a particular processor will be reallocated on the same processor. Failure to account for this effect can result in memory blocks being reallocated on different processors for every allocation, causing excessive cache coherence traffic.

However, the danger with using per-processor freelists is that memory is wasted if it is left on free lists for processors that have no need for the memory. Hence, it is important to return the memory to a more central pool periodically to allow other processors to use the memory if their free lists run low.[9]

False sharing

Even if one manages to achieve a high degree of multiprocessor locality, false sharing can quickly erase any potential gains. False sharing occurs when two pieces of data that are accessed independently by different processors happen to reside on the same cache line. With large cache lines (as is common on today's multiprocessors) and many small objects (common in object oriented systems) this can be a serious problem.

False sharing can occur in two primary ways in a memory allocator. First, it can occur when multiple small chunks of memory within a single cache line end up allocated to objects that are accessed on different processors. This can happen even with per-processor free lists if the sub-cache-line objects are allocated from one processor and accessed on another. Second, false sharing can occur in the data structures used by the memory allocator itself.

One solution to reduce false sharing is to make the minimum allocation unit the same size as the cacheline. However, with cachelines of 64 or 128 bytes this can result in unacceptably high internal fragmentation. Another approach is to assign each cacheline-sized block of memory a home processor, similar to the home node of memory in a NUMA multiprocessor, and always return memory to the home processor as soon as it is freed. Unfortunately, this would require a high overhead for small blocks, something we would like to avoid. A final approach is to provide separate free lists for allocating and freeing memory that is known to be accessed strictly locally. This requires the client

[9] This is similar to the need to coalesce free memory blocks in regular memory allocators to allow the memory to be reused as different sized blocks.

to know which list to allocate and free from, but provides the possibility of improved performance when the situation is identifiable.

Cache reuse

Although the memory allocation subsystem must, for good performance, take into account the multiprocessor caching issues described above, it is also important that it maximize cache reuse on a per-processor basis (again, because of the high cost of cache misses and the large secondary caches). The most effective way to maximize temporal locality is to reuse free blocks soon after they are freed, when they are likely still in the cache. To increase spatial locality, if possible, the allocator should allocate small memory chunks from the same cache line in quick succession on the assumption that the memory is likely to be used close in time, maximizing the use of the large cache lines.

Concurrency control

Maximizing concurrency in a multiprocessor is probably the second most important component of a high performance allocator, after cache efficiency. As was discussed in Chapter 2, per-processor data structures help increase not only locality, but concurrency as well. As with most data structures under Tornado, we are primarily concerned with designing for the expected common case of low contention.

Memory allocation would be an ideal facility in which to apply lock-free approaches to concurrency control. Unfortunately, the primitives provided on most current architectures are not sufficiently powerful to apply most of the techniques that have been developed. For example, most processors disallow any loads or stores between a load-linked and store-conditional instruction, which makes it exceedingly difficult to implement even the most basic linked-list in a lock-free manner (see previous section on locking).

Exception level support

Because the allocator needs to work in a kernel environment, it must be capable of dealing with exception-level synchronization issues. The allocator must ensure that deadlocks cannot occur when exception-level operations make use of the memory allocation facility. Three approaches are generally used to achieve this: (i) disabling interrupts (in addition to locking) at key points in the allocator; (ii) restricting the types of memory allocation operations exception-level code is allowed to invoke; and (iii) using lock-free synchronization structures whenever possible. We have already ruled out the third option (above), leaving disabling interrupts and adding restrictions to exception-level memory allocation code. Since disabling interrupts for long durations is generally inadvisable, and since memory allocation at exception-level can always fail for other reasons (such as the need to do page out to reclaim physical memory), the approach we are left with is adding restrictions to the exception level code. Our approach is to provide a best-effort interface that may fail to allocate memory if the necessary locks are not currently available.

NUMA awareness

Because our target architecture is a large-scale NUMA multiprocessor, the allocator must have some awareness of the differing costs of memory and attempt to ensure that memory allocated from a given processor is located on the nearest available memory module. This also means that when freeing memory the system must attempt to return it to the home free list. (These issues are similar to those that concern maximizing cache locality).

There are various tradeoffs in terms of when and how the system should check for the home node of freed memory. For instance, checking on every free operation will ensure that memory is always returned to the right place, but adds extra overhead to each free operation. In addition, the memory might be in the processor's cache, meaning that future allocations of the block might best be made from the processor that just freed it, despite the home node being remote. On the other hand, delaying the check could mean loss of locality and increased traffic due to excessive remote accesses.

Internal/external fragmentation

Efficient use of memory remains an important issue for memory allocators. Even with large amounts of physical memory, wasted memory can cause excessive cache misses and paging (depending on the distribution of the unused chunks). Wastage in memory allocators generally arises from four main sources: $(\boldsymbol{i})$ internal fragmentation caused by allocating a block larger than requested; $(\boldsymbol{ii})$ external fragmentation caused by ignoring free blocks because they are not of the right size; $(\boldsymbol{iii})$ free-list fragmentation caused by ignoring free blocks because they are on the "wrong" free list; and $(\boldsymbol{iv})$ overhead, caused by the use of boundary tags that extend the size of the requested block. These issues have been well covered in the literature [Wilson *et al.*, 1995], with the conclusion that most memory allocation strategies can achieve low wastage, but at the potential cost of high-latency and low-locality. It is therefore a question of determining the desired tradeoff, once the particular allocation strategy has been chosen. The issues will become more clear when we examine memory allocation implementations in more detail.

Execution time

Execution time can usually be minimized by using free lists of different sizes (to speed up the time to find a block of a particular size) and deferring free-block coalescing to allow rapid reuse of common block sizes and hopefully avoiding the cost of coalescing altogether.

Return unused memory

Finally, because the kernel and servers run continuously, it is important that they be able to return memory that was allocated during peak demands back to the general system pool, so that it can be reused by other programs. Many of the current user-level schemes do not support returning memory back to the system, but instead retain all free memory in the allocator's private pool. Those that do return memory to the pool often restrict the memory that can be freed to blocks that are at the very end of the heap. However, we require an allocator that has the ability to return arbitrary free pages back to the general system pool, both for the kernel and user-level servers, because otherwise there is a tendency for total memory requirements to continually grow over time.

Conflicting goals

Unfortunately, several of the goals listed above conflict with one another. For example, many of the techniques for improving performance in multiprocessors require the use of per-processor free lists. This can increase memory wastage and inhibit the ability of the allocator to return unused memory, due to the fragmentation of memory across the various lists. Similar wastage also occurs with the use of per-block-size free lists. As discussed above, NUMA support generally requires extra checks for the remote-memory case, increasing the base execution time, and can conflict with cache reuse when recently accessed remote memory is freed locally. Choosing the right tradeoffs therefore requires careful attention to the requirements of the particular target environment.

6.2.2 Tornado allocation facility

Background

As a starting point, we decided to base the Tornado allocation facility on the allocator by McKenney and Slingwine [McKenney and Slingwine, 1993] as it provides features closest to what we require. It was designed for shared-memory multiprocessor kernels, and expends significant effort to maximize multiprocessor locality, reuse, and concurrency. Although close to meeting our requirements, the McKenney and Slingwine allocator still requires a number of extensions and modifications in order to fit our system. Their hardware platform was a Sequent 2000 with relatively slow i486 processors running at 25MHz. This is a bus based multiprocessor, and hence only cache locality considerations are necessary (i.e., all memory is equally distant from the processors' point of view). Reducing lock overhead was a major factor in their design, necessitated by the fact that each synchronization operation required a shared-bus transaction. Also, their relatively small cache lines (32 bytes) did not require special attention to false sharing.

The primary issues that need to be addressed are therefore: $(\boldsymbol{i})$ dealing effectively with large caches and large cache lines, $(\boldsymbol{ii})$ adding support for user-level system servers, $(\boldsymbol{iii})$ including support for NUMAness, and $(\boldsymbol{iv})$ adapting the design to use the now-pervasive load-linked/store-conditional synchronization primitives.

Overview

The basic design presented by McKenney and Slingwine consists of four layers, with the common cases handled by the lower layers that are designed to be fast and highly concurrent, and the less common cases that require more cooperation and less speed handled by the upper layers. The four layers are:

1. a per-processor layer that keeps independent free lists of memory blocks on each processor, for high locality, cache hit rate, and concurrency;

2. a global-layer, that allows blocks freed on one processor to be eventually re-allocated by another processor;

3. a coalesce-to-page layer, that manages pages of memory and keeps track of outstanding memory blocks for each page allocated to the lower layers; and

4. a virtual memory layer, that keeps track of free regions in the virtual address space, for mapping new pages when they are required by the coalesce-to-page layer.

For Tornado, we keep the basic four layer design. However, in order to provide NUMA locality, we replicate all but the per-processor layer (which is already replicated), with one instance per cluster of processors. This increases locality at the cost of some increased complexity. The details of how this affects the design will be discussed in the context of each individual layer.

Per-processor layer

The per-processor layer is based on an array of free-lists (see Figure 6.1), one for each fixed block size supported by the system. In the original design, power-of-two sized blocks were used, from

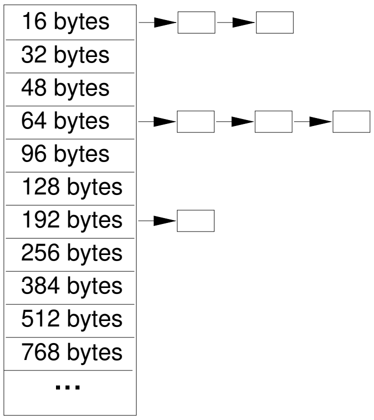


Figure 6.1: *Array of free lists, one for each supported block-size.*

some minimum size, say 16 bytes, up to some maximum size, say 4096 bytes. Larger requests use the virtual memory layer directly. A request for a given block size is satisfied by rounding the size up to the next power-of-two size, finding the appropriate free list, and removing the first element. This may waste a significant amount of space (up to 50%), but is very fast. Because of the space waste, the Tornado implementation supports arbitrary (double-word aligned) sized blocks, while still supporting fast common-case allocation.

When allocating a block, if the size is known at compile time, the head of the free list can be computed at compile time, making allocation particularly fast. If the size is not known at compile time, but is guaranteed to be less than the maximum size supported by the free lists, the size is used to index into an array of pointers to the free lists. If nothing is known about the size at compile time, a check is made at run time to determine whether the allocation should use the free lists or go directly to the virtual memory layer. It is up to the user of the system to choose the appropriate allocation interface based on the above conditions.

Freeing a block requires determining which free list the block came from. Again, if the size is known at compile time, this is trivial. Otherwise, the system uses the coalesce-to-page layer to determine the size of the block and hence indirectly find the free list.

In order to prevent the free lists from becoming too long and wasting memory, several options are available:

- Have a scavenging daemon scan all the lists and steal back excess memory periodically;
- If an allocation on one processor fails to find a free block, have it scan its neighbours;
- Keep a count of the number of elements in each free list and have the processor return memory to the higher layers itself when it pushes past the limit.

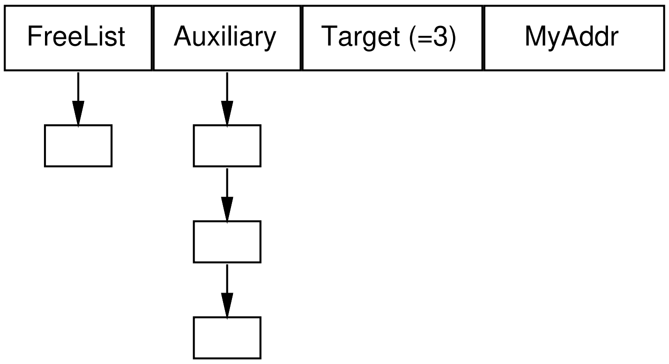


Figure 6.2: *This figure shows the per-processor free list header structure, with a target free list size of 3 elements (the auxiliary list will always be either empty or have* Target *number of elements).*

The original McKenney and Slingwine design chose the last option, and we do the same. This design uses an additional auxiliary list associated with each free list (see Figure 6.2). When the free list length reaches the target length, the entire free list is moved to the auxiliary list, before the newly freed element is put on the free list. If the auxiliary list is already full, it is first moved up to the next layer. Similarly, if during allocation the free list is found to be empty, the auxiliary list is consulted, and if not empty is moved to the free list. Otherwise the free list is replenished from the next higher layer. This design limits the amount of memory held in the lowest layer and prevents the free list from continually bouncing between the layers.

The use of per-processor freelists gives each processor a private set of freelists from which to allocate and deallocate memory (see Figure 6.3). Since they are private, they experience no contention from the other processors, and since most operations can be satisfied by the per-processor free lists, high locality and good concurrency should be achievable. The one drawback to this approach is that memory on one processor's free list is not available to the other processors, and hence the total memory requirement of the system may grow.

In the original design, the free lists were protected by disabling interrupts since locks were expensive on their architecture and the allocator was only intended for the kernel. Ideally we would use a lock-free implementation for the free lists, but as was discussed earlier, this is complicated by the limitations of the atomic primitives.

We chose instead to use a more traditional locking approach, but customized for the particular requirements of the allocator. This choice, however, constrains exception level allocation and deallocation in that the software must deal with the possibility that allocations can fail if the free list is empty or locked (similarly for deallocations). Although this might impose greater limitations on allocations and deallocations at exception level compared with other approaches, the other approaches must also be prepared to handle allocation failure at exception level, since paging may be

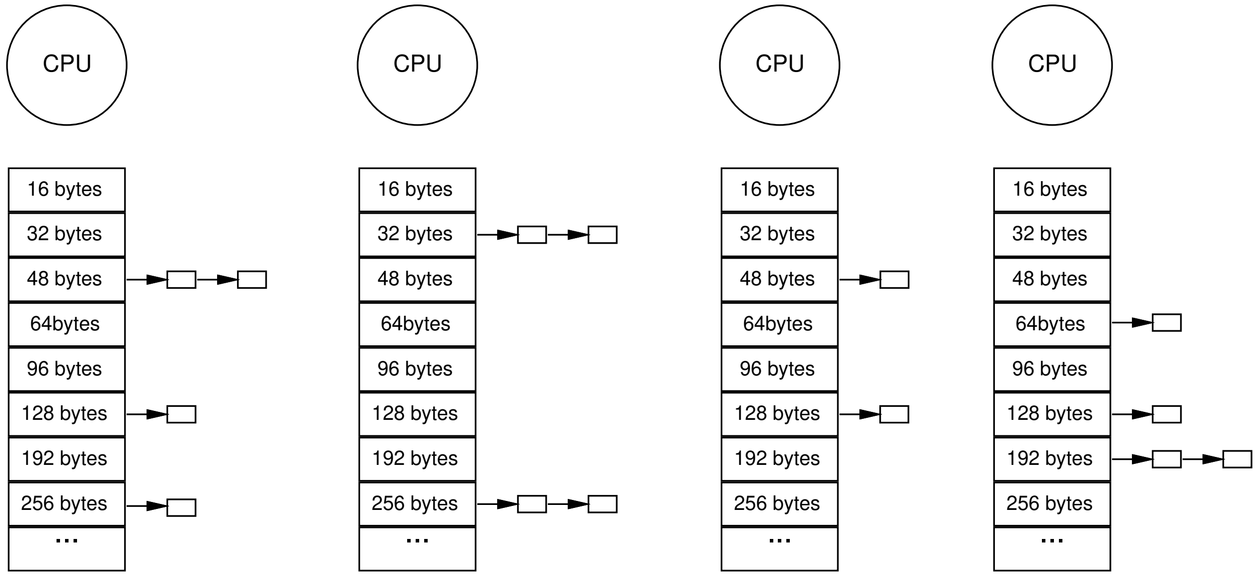


Figure 6.3: *Free list per processor arrangement.*

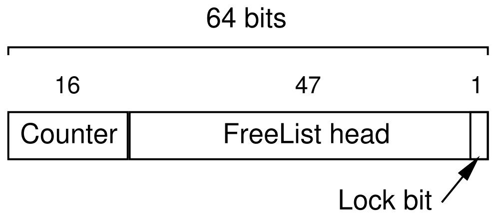


Figure 6.4: *This figure illustrates how the head pointer for the free list combines a counter, a lock, and free list pointer in a single 64 bit word.*

needed to free up enough space.

To avoid the traditional overhead of locking, we take advantage of the load-linked/store-conditional instructions to build a specialized support routine for the allocator. Because of the limitations of these instructions, it was necessary to combine a number of key fields into a single 64 bit word. This includes the head of the list, the count of the number of elements in the list, and a lock bit (see 6.4).

On allocation, we use the load-linked instruction to load the combined word, then check the lock bit and the freelist head field to see if it is safe to allocate. If it is, a store-conditional is used to lock the free list. If the store-conditional fails or if the above check failed, we call a more general purpose routine that handles exceptional cases. If the store succeeds, we remove the top element from the list and store the pointer to the next element back in the head. This pointer already has the counter field set to the size of the remaining free list and has the lock bit clear. Because the lock bit is clear, storing the new head effectively unlocks the list.

On deallocation, we check whether the memory is local, attempt to lock the list, and check the length of the list against the maximum length, again passing the call on to a more general purpose routine if any of these checks fail. Assuming they all succeed, the previous head is stored in the new element, the list length is incremented and combined with the pointer to the new element and stored as the new head, effectively releasing the lock.

If the memory is from a remote cluster, a number of options are available. One option would be to treat it like local memory (and eliminate the check above). Not only would this simplify the common case, but reallocating the memory to the current processor might make sense since it is likely in the current processor's cache. However, in the long run, it may be costly, as all NUMA locality could be lost. It would also make optimizations that depend on receiving local memory from the allocator more difficult.

A second option would be to free the memory to the home node. This would ensure that all local allocations always return local memory, but it could be costly because of all the remote accesses incurred for every remote block that is freed.

We chose a third option, as a compromise between the two alternatives above. Instead of freeing remote blocks back to the home cluster, we instead put them on a local list for remote blocks from the target cluster. When the list reaches the target length, the whole list is moved to the Global layer of the home cluster. With this approach, when the list is being built, the memory being touched is likely already in the cache, and when the list is moved back to the home node, the cost of the remote access is amortized over the entire length of the list. However, there is a tradeoff in the use of more free lists: more memory is fragmented among these lists, and the number of lists per processor grows with the size of system. Some of the problems can be alleviated by combining lists for more distant remote processors, and using a daemon that periodically scans the lists and returns free memory to ensure it doesn't remain on these lists indefinitely. However, more experimentation is required to determine if such measures are warranted.

In order to ensure the check for remote frees on deallocation is efficient, we divide the virtual address space dedicated to the allocator into separate chunks dedicated to each cluster and store the identifying upper bits of the local cluster in the local freelist header. This allows a simple mask and compare to determine if the memory is local or not.

Global layer

The global layer's sole purpose is to redistribute free blocks from heavy deallocators back to heavy allocators. The structure used at the global layer is similar to the per-processor structure in that it has an array of headers for each block size. The global layer, however, maintains for each block size a list of lists, where each sublist matches the size of the per-processor Target list size (see Fig-

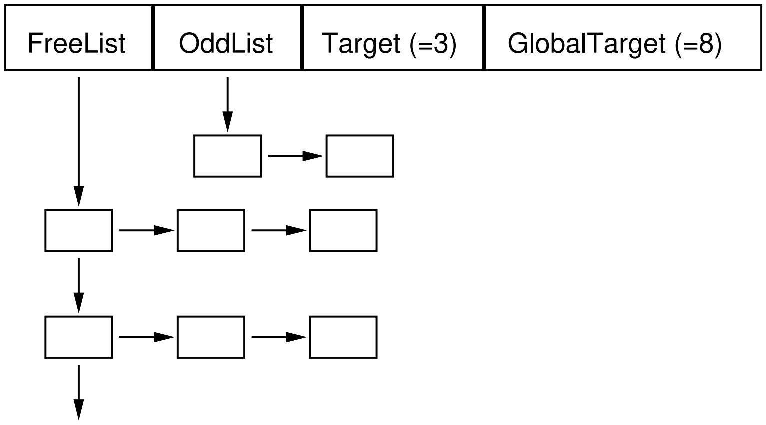


Figure 6.5: *This shows the global layer free list structure. Each of the sublists hanging off of* FreeList *are* Target *number of elements long.* OddList *holds left-over elements. There can be a maximum of double* GlobalTarget *number of sublists before* GlobalTarget *of the sublists must be moved to the next layer above.*

ure 6.5). Like the per-processor layer, the global layer has a target value for the number of sublists it wants to keep. When twice that value has been reached, half the lists are moved up to the next layer for coalescing back into pages. When there are no more sublists and an allocation request comes from the per-processor layer, the global layer fills its lists from the layer above. As with the per-processor layer, this organization is intended to reduce the frequency with which lists need to be moved between layers.

Since all the sublists must be the same length, namely the same as the target length of the per-processor lists, any left-over elements are put on a separate *odd list*. (These left-over elements sometimes come from the layer above, when a request can not be satisfied exactly.) When this list reaches the required size it is added as a sublist to the list of sublists.

Unlike in the original design, which was targeting a bus-based multiprocessor, our allocator has one global layer for each cluster of processors (i.e., for each neighbourhood of processors and memory modules), rather than one for the entire system. This facilitates sharing within the cluster (where we expect most interactions to occur) while improving NUMA locality and concurrency within the global layer.

Coalesce-to-page layer

The coalesce-to-page layer has a number of purposes, centered around the pages of memory from which blocks have been allocated. It acts as a source and sink of memory blocks from the global layer. Its purpose is to coalesce blocks into pages until complete pages can be freed, and to request whole pages from the layer above to be partitioned and given to the lower layers when needed.

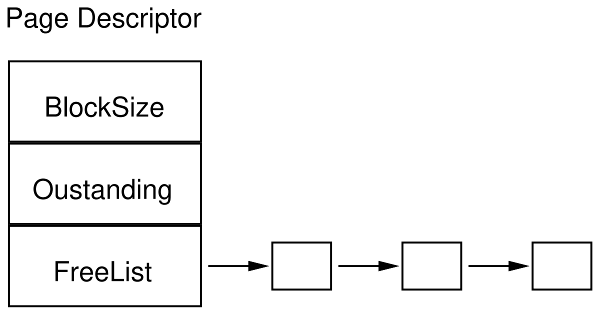


Figure 6.6: *Memory allocator page descriptor.*

In the kernel, the pages are physical pages allocated from the global pool of free memory, but in system servers, the pages are virtual memory pages. In both cases, the pages are treated the same: they are allocated when more memory is needed, and the memory (backing them) is released when all blocks of memory on the page have been freed and coalesced. Also, in both cases, there is one instance of this layer for each cluster (i.e., there is a one-to-one relationship between global layer instances and coalesce-to-page layer instances).

Each page in the coalesce-to-page layer contains only blocks of the same fixed size. If there are no pages with free blocks of the given size, a new page is requested from the virtual memory layer, and split up into a list of blocks of the requested size.

The coalesce-to-page layer is also responsible for keeping track of the status of each page that has been allocated to it. The information includes the size of the blocks in the page, how many blocks have been handed down to lower levels (so that it knows when the page can be given back to the virtual layer), and a list of all the blocks in the page that are available for future requests from the lower layers. The information is maintained in a descriptor which can be efficiently located given the page's address.[10] The descriptors are also used directly by the lower layers to determine the size of the block given only its address.

Virtual layer

This layer is responsible for managing the virtual address space for user-level servers, and for interfacing with the system page allocator in the case of the kernel. It handles requests for more pages from the coalesce-to-page layer as well as large size (multi-page) memory block requests from any

[10]The mapping is done using a simple array of descriptors that precedes each chunk of virtual memory in use, for the user-level system server case, and with a hash table for the kernel. The reason for using different approaches for the servers and kernel is that since the kernel uses physical memory directly, it can't ensure that all allocated memory will be nicely compacted into contiguous regions. This would require that an array of descriptors be large enough to cover all of memory, which would be prohibitive.

of the layers below (these requests bypass the layers in between, since the purpose of the lower layers is only to deal with small size blocks that need to be coalesced and managed as lists).

As with the previous two layers, there is one virtual layer per cluster of processors.

Reducing false sharing

One key issue we have not yet addressed is how to reduce false sharing. The approach we take is to effectively provide a separate set of the first three layers for allocations of memory where it is known a priori that the allocated memory will only be used on the local processor. (Actually, we only need to provide separate layers for block sizes smaller than a cacheline). This segregates small blocks that are accessed locally from all other blocks, ensuring that they can't end up on a cacheline that is shared across processors, but it requires the clients to use the proper interface if they wish to optimize for locality.

6.2.3 Performance evaluation

Unfortunately, some of the NUMA aspects of the Tornado memory allocator are still incomplete, so we present results with a single shared global layer, coalesce-to-page layer, and virtual layer. As a result, scalability beyond requests to the lowest layer is limited.

Table 6.2 shows the allocation and deallocation costs for two different cases. In the first case, *best-case* results are obtained by measuring the time to allocate and then deallocate a single block in a tight loop. This ensures that all allocation and deallocation requests are handled by the lowest layer. For this case, allocation requires 16 instructions and deallocation 21. Included in this cost are the various checks and computations required to keep track of the length of the freelist and to determine if it is over-full, which cost four instructions in the case of allocation and eight instructions in the case of deallocation. Checking whether remote data is being deallocated cost an additional three instructions to the deallocation cost (also included in the numbers presented in Table 6.2).[11] Although these extra costs appear significant, most of this work can be performed concurrently with the critical path work. As a result, the costs on the R10000 processor of a minimal allocator with none of the checks and the Tornado allocator are roughly the same.[12]

The second case considers a scenario with close to worst-case behaviour, where a stream of allocations is followed by a stream of deallocations. The test involved a stream of 32768 allocations and deallocations of 256 byte blocks, enough to stress all levels of the allocator. About 80 percent of

[11] These results include the NUMA check even though only a single upper layer is used.

[12] The results actually show the Tornado allocator doing better. This is most likely due to different alignments of instructions. Repeating the experiment with different code layouts changed the times by a few cycles, but the relative ranking always remained the same.

Operation	Instructions	Cycles
alloc (best-case)	16	(total)
dealloc (best-case)	21	(49)
alloc (stream-case)	75	668
dealloc (stream-case)	156	690

Operation	R10000
minimal	10
Tornado	7

Table 6.2: *The table on the left includes time in instructions and cycles for NUMAchine, for best-case allocations and deallocation when they can be satisfied from the lowest layer, and for the case when multiple requests are issued in succession, first for allocation and then for deallocation, causing worst case behaviour. For the alloc/dealloc best-case test, it was not possible to accurately measure the individual costs on NUMAchine, so only the total is presented for those two. The table on the right compares the number of cycles on an R10000-based SGI system for a minimal allocation system and for Tornado's allocator.*

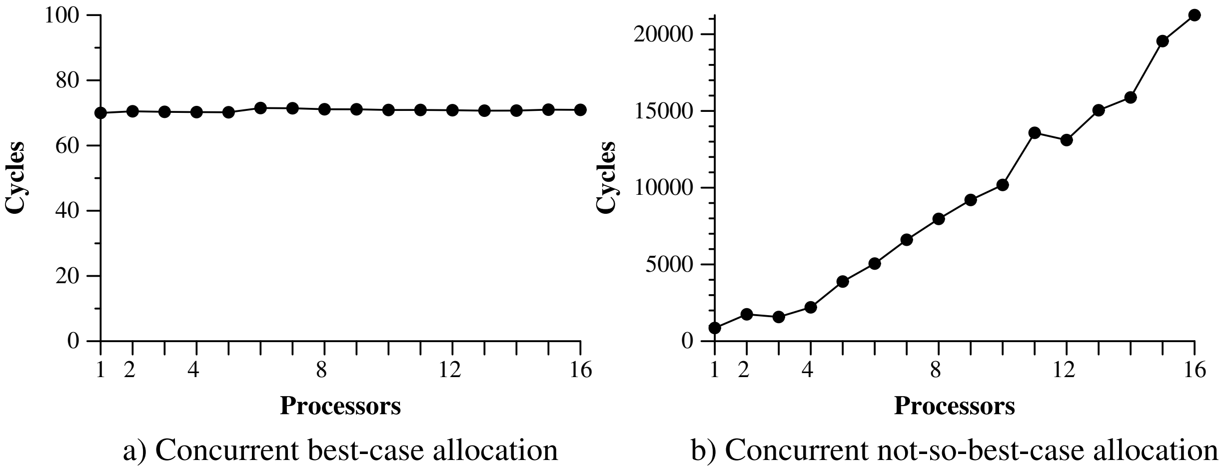


Figure 6.7: *Concurrent test for the best-case and not-so-best-case allocation tests.*

the time for allocation is spent in the coalesce-to-page layer, while about 50 percent for deallocation is spent mapping the block address to the memory descriptor, which is only required for the kernel version due to special constraints in the kernel. These are two areas that have not received as much attention in optimizing the code, and it shows. However, this also suggests that performance can likely be improved significantly for such stressful situations with more tuning.

Finally, Figure 6.7 shows concurrent versions of the tests above. As expected, the best-case concurrent test scales perfectly, with no slowdown effects as more processes are added. For the case with a stream of requests, performance degrades rapidly. However, for up to 4 processors (the size of a NUMAchine station), performance degrades more slowly, suggesting that the NUMA design would likely be effective in supporting scalability even under significant stress.

6.2.4 Open issues

As with most components of Tornado, there are still many issues that remain to be addressed:

- Freeing remote data: as indicated earlier, there are many options for how to handle freeing remote data, each with performance advantages under different usage patterns. Determining whether our selected approach is sufficiently efficient and robust will require a more extensive study of actual dynamic allocation usage.

- Memory wastage: there is a danger of significant wastage of memory, both within the memory blocks due to the small number of block sizes supported, and in the various lists due to the dispersion of free blocks to a large number of free lists within and across processors; a proper study of the real amount of wasted memory in a running multiprocessor system is needed.

- Closer integration with C++: the allocator is most efficient if the size of the object being allocated is known at compile-time; this is sometimes awkward to program in C++ and could easily be automated since the compiler knows at each **new** location the size of the object being created.

- Proper evaluation: given the issues above and the complexities of a full running system, proper evaluation considering the kernel and servers on a large number of processors running a real workload would be highly beneficial in evaluating just how effective the design is in limiting contention, increasing locality, and minimizing memory wastage.

6.3 Related Work

Our dynamic memory allocation design borrows heavily from the work of McKenney and Slingwine [McKenney and Slingwine, 1993], which is one of the few published works on multiprocessor memory allocation, in particular for kernel environments. A survey paper by Wilson [Wilson *et al.*, 1995] covers many of the other schemes, but does not address multiprocessor or caching issues. Grunwald et al examined cache performance of allocation schemes [Grunwald *et al.*, 1993] and suggest a number of techniques they felt would be most effective in dealing with locality issues. Most of these techniques can be found in the McKenney and Slingwine memory allocator (with a few additions in our own adaptation).

There have been a number of allocators designed specifically for operating system software. A good overview can be found in [Vahalia, 1996]. Most share a similar desire to reduce allocation and deallocation to simple list manipulations, and to provide an efficient way to coalesce free blocks back into free pages that can be returned to the general free-memory pool. However, apart from the work of McKenney and Slingwine, none of the allocators described deal with multiprocessor issues, and none, including McKenney and Slingwine's allocator, consider the unique problems faced in a NUMA multiprocessor, nor do they consider such issues as false sharing or user-level support.

6.4 Summary

In this chapter, we looked at the key design and implementation issues of two of the most important base components of a multiprocessor operating system: the locking facility and the dynamic memory allocation facility.

In Tornado, the prime concern of the locking facility is efficiency in the uncontended case, both in terms of time and space. Focusing on the uncontended case makes sense because the clustered object system should help keep contention for any one clustered object representative relatively low. It is, of course, still important to degrade as gracefully as possible under heavy load, but beyond that, there is little that can be done under such circumstances.

The dynamic memory allocator faces a number of demands. It must be efficient in time and space, maximize cache reuse and NUMA locality, coalesce free memory promptly, and be highly concurrent. The Tornado allocator attempts to address all of these issues, with varying degrees of success. The use of per-processor structures and separate freelists for strictly local allocations addresses many of the issues, with further divisions along cluster lines helping to address the NUMA issues. However, further analysis is required before one can conclude whether the right tradeoffs have been made.

Chapter 7

Overall System Performance

To evaluate the effectiveness of the Tornado design at a level above the individual components, we ran a few multiprocessor operating system stress tests. The micro-benchmarks are composed of three separate tests: thread creation, in-core page faults, and file stat, each with n worker threads performing the operation being tested concurrently. For comparison purposes, we ran the same tests on a number of Unix systems.

Thread Creation Each worker successively creates and then joins with a child thread (the child does nothing but exit). Pthreads are used for the Unix systems; regular Tornado processes are used for Tornado.

Page Fault Each worker thread accesses a set of in-core unmapped pages in independent (separate mmap) memory regions.

File Stat Each worker thread repeatedly fstats an independent file.

Each test was run in two different ways; multi-threaded and multi-programmed. In the multi-threaded case, the test was run as described above. In the multi-programmed tests, n instances of the test were started with one worker thread per instance. In all cases, the tests were run multiple times in succession and the results were collect after a steady state was reached. Although there was still a high variability from run to run and between the different threads within a run, the overall trend was consistent.

Figure 7.1(a) shows normalized results for the different tests on NUMAchine. Because all results are normalized against the uniprocessor tests, an ideal result would be a set of perfectly flat lines at 1. Overall, the results demonstrate good performance, since the slowdown is usually less than 50 percent with up to 16 processors. However, there is high variability in the results, which accounts for the apparent randomness in the graphs.

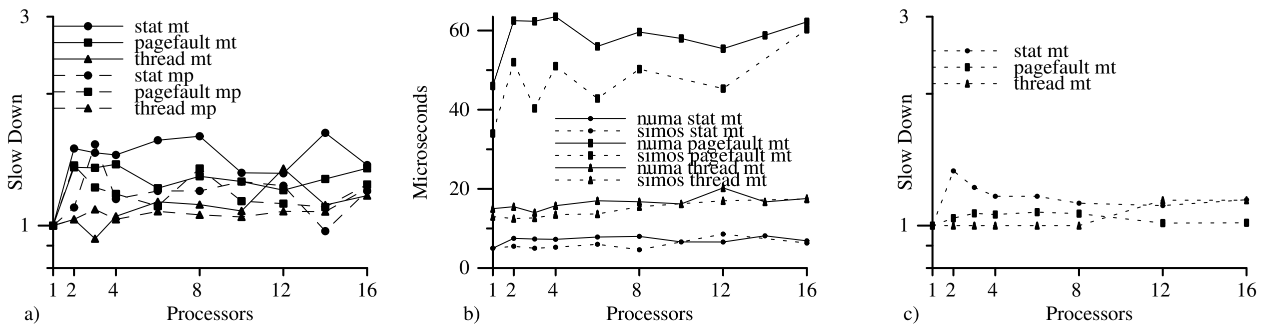


Figure 7.1: *Micro-benchmarks: Cost of thread creation/destruction (thread), in-core page fault handling (pagefault), and file stat (stat) with n worker threads running either in one process (mt) or in n processes with one thread per process (mp). The left figure (a) shows slowdown relative to the uniprocessor case. The middle figure (b) shows the raw times in microseconds for the multithreaded tests on NUMAchine and SimOS, and the right figure (c) shows the slowdown on a log scale of the multithreaded tests run on SimOS configured with 4-way set associative caches.*

System	OS	# Cpus	Legend
UofT NUMAchine	Tornado	16	numa
SimOS NUMAchine	Tornado	16	simos
SimOS 4way NUMAchine[a]	Tornado	16	simos4way
SUN 450 UltraSparc II	Solaris 2.5.1	4	sun
IBM 7012-G30 PowerPC 604	AIX 4.2.0.0	4	ibm
SGI Origin 2000	IRIX 6.4	40[b]	sgi
Convex SPP-1600	SPP-UX 4.2	32[c]	convex

[a]simulated 4way set associative
[b]Maximum used in experiments is 16
[c]Maximum used in experiments is 8

Table 7.1: *Platforms on which the micro-benchmarks where run.*

Similar results are obtained under SimOS. Figure 7.1(b) shows the raw times in microseconds for the multi-threaded tests run under NUMAchine and SimOS. As we saw in earlier tests, setting SimOS to simulate 4-way associative caches smoothes out the results considerably, as shown in Figure 7.1(c).[1]

To see how this compares to the performance of existing systems, we ran the same tests on a number of systems available to us (see Tables 7.1 and 7.2 and Figure 7.2). The results demonstrate a number of things. First, many of the systems do quite well on the multiprogrammed tests, reflecting the effort that has gone into improving multi-user throughput over the last 10-15 years. However, the results are quite mixed for the multithreaded tests, being at times over a factor of 100 slower

[1]There is still an anomaly involving a single thread taking longer than the others that we have yet to track down. This pulls up the average at two processors, but has less effect on the average when there are more threads running.

Operation	Thread Creation	Page Fault	File Stat
NUMAchine	15	46	5
Sun	178	19	3
IBM	691	43	3
SGI	11	21	2
Convex	84	56	5

Table 7.2: *Base costs, in microseconds, for thread creation, page fault handling, and file stating, with a single processor.*

with 16 processors. In particular, although SGI does extremely well on the multiprogrammed tests, it does quite poorly on the multithreaded tests. This is particularly interesting when compared to results on an older bus-based system (a 6-processor SGI Challenge running IRIX 6.2, results not shown), where the multiprogrammed results are slightly worse and the multi-threaded results are quite a bit better. This appears to reflect the strength and direction of the new NUMA SGI systems. Overall, Sun performs quite well with good multithreaded results and respectable multiprogrammed results; however, we had only a four processor system available, so it is hard to extrapolate the results to larger systems. Surprisingly, IBM does quite poorly across all of the multithreaded tests and a couple of the multiprogrammed ones, while Convex does poorly for the thread creation and page fault test in both multi-programmed and multi-threaded mode.

Overall, the results of Tornado compared with the other systems demonstrate the strength of the Tornado approach, but also point to the need for greater attention to cache conflict issues (although more recent processors with their associative caches tend to address this problem), which can be particularly difficult to address in dynamic object-oriented systems like Tornado.

However, it is important to keep in mind that these results can reflect many things about a system beyond the level of concurrency supported or the maturity of the multiprocessor implementation. For example, the use of a single shared ready queue limits scalability and locality, but supports finer-grained load balancing. In Tornado, we have primarily focussed our early development on concurrency and locality, at the expense of load balancing. As a result, it performs well for such well balanced tests as those above. This tradeoff between locality and resource sharing, both in the implementation and the policies supported, is an important issue, but beyond the scope of this dissertation.

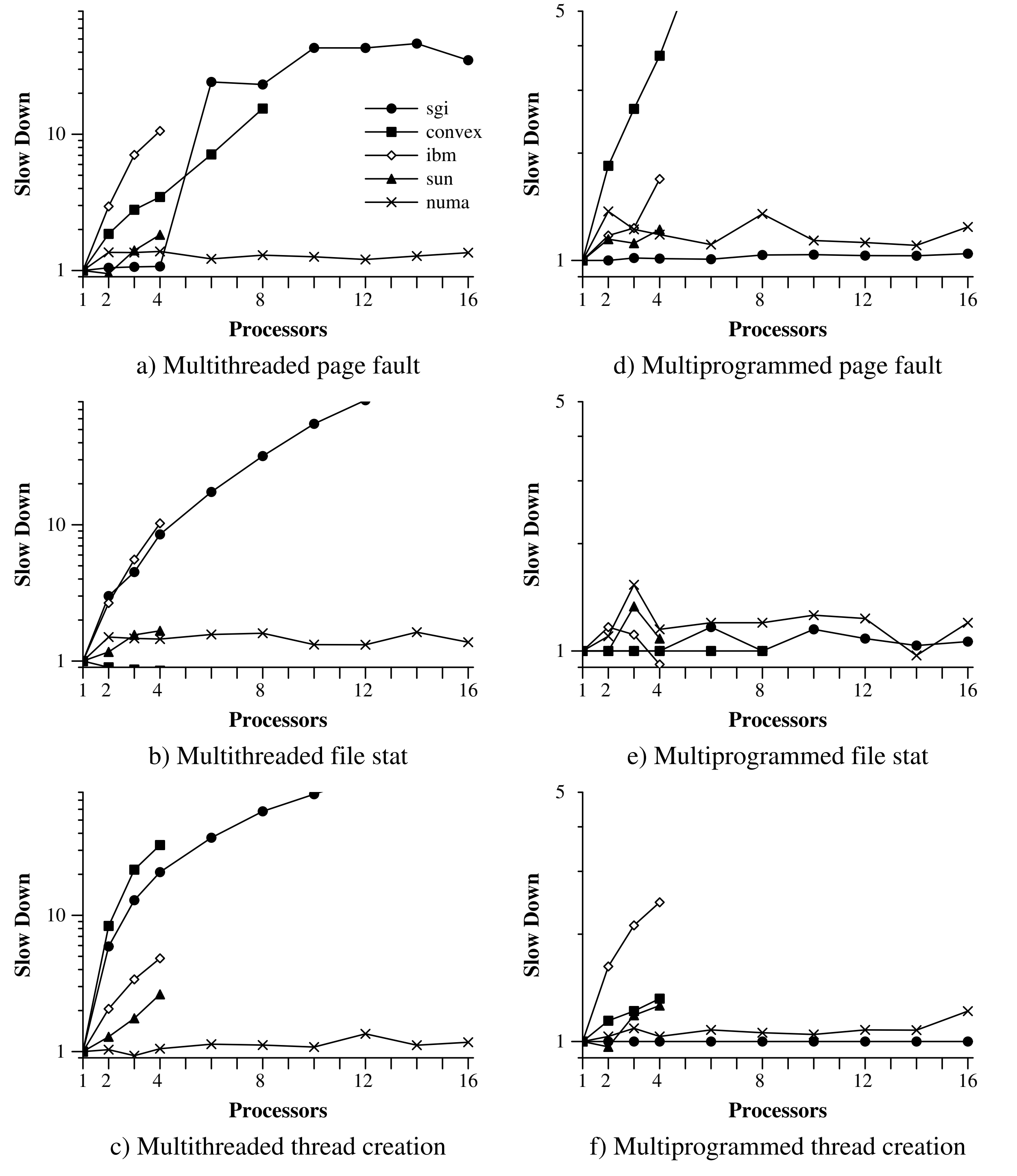


Figure 7.2: *Micro-benchmarks across all tests and systems. The systems on which the tests were run are: SGI Origin 2000 running IRIX 6.4, Convex SPP-1600 running SPP-UX 4.2, IBM 7012-G30 PowerPC 604 running AIX 4.2.0.0, Sun 450 UltraSparc II running Solaris 2.5.1, and NUMAchine running Tornado.*

Chapter 8

Overall Summary

In this dissertation we have presented a number of contributions that are the result of our research. Although the implementation is only partially complete and open issues remain to be addressed, we believe the basic premise of the work and the conclusions drawn from it are valid. In this chapter, we consider the main contributions of our work and the impact the open issues may potentially have on them.

8.1 Object-Oriented Structure

The object-oriented structure of Tornado encourages all virtual and physical resources to be represented by independent objects, providing a framework for locality, concurrency, and flexibility.

Benefits.

- Because each resource is managed by an independent object, contention for locks or other resources is limited to those components that are truly shared by more than one program, significantly reducing contention. For example, the cache memory for separate files is managed by independent FCMs ensuring that accesses to the two files never contend for FCM page cache locks.
- The independent objects can be located near the clients accessing it, promoting increased locality.
- The object-oriented structure allows different implementations of the same component to be supported, allowing the system to more closely match the implementation and policy requirements of the applications. For example, different implementations of the Program object, one for sequential programs and one for parallel programs, can be provided.

- Applications can choose at run time the particular implementation of any component they desire, allowing them more control over how their resources are managed. This allows, for example, an application to choose a particular cache management strategy for a particular file.

Open Issues.

- Space inefficiencies may result from localizing data structures within objects. For example, the overheads of complex data structures are higher when they manage smaller amounts of data.

  In most cases this can be addressed by using alternate data structures, such as dynamically sized hash tables instead of static ones. In addition, there can be significant time savings, since most data structures are more efficient to access when they hold fewer elements.

- Partitioned management of resources makes it more difficult to implement global policies. For example, providing a global least-recently-used page replacement policy is more challenging when the pages are managed by multiple independent page caches.

  Although implementing such global policies requires more careful consideration of the information sharing interfaces of the objects, in most cases it can be done [Wilk, 1997]. As a general trend, most systems are moving toward more localized resource management policies in order to provide differentiated services, which our structure actually facilitates.

8.2 Clustered Objects

Clustered Objects provides a framework for distributing the state of an object to improve locality and concurrency in a manner transparent to its client objects.

Benefits.

- Clustered objects enhances locality and concurrency by distributing object state, supporting migration, replication, and object partitioning. For example, replicating a read-only object provides local copies for all client objects and eliminates the bottleneck a single copy would have.

- Overhead to access the appropriate object rep is kept low by using per-processor object translation tables that are located at the same virtual address on every processor. This allows a single extra pointer indirection to suffice for accessing a local rep.

- Because the clustered object translation entry for a given object is located at the same address on all processors, a common pointer (to the translation entry) can be used as the reference to the clustered object on any processor in the system. This ensures that the distribution of object state can remain transparent to the client.

Open Issues.

- Page replacement for the physical memory backing the object translation table has not yet been implemented. It is therefore unclear how effective such a strategy can be, or whether the use of a compression table might be necessary.

  Paging the translation table could be inefficient if reps are migrated frequently or there is little locality between the processors creating the clustered objects and the processors accessing them. This would cause references to be scattered throughout the table, and threaten the viability of a single linear clustered object translation table. Alternate data structures are possible, but would add to the common case cost of a translation table hit. However, we expect there to be a fair degree of locality in the object reference pattern for the majority of the objects, based on a mix of large-scale parallel and sequential workloads.

- A complete strategy for supporting the migration of clustered object reps has not yet been developed. Migration is complicated by the need to destructively copy an object while it is being accessed. The problem is similar to that faced by copying garbage collectors in object-oriented language systems, but without the language support these systems depend on.

  Although strategies based on the same approach currently used for clustered object garbage collection are being considered, many details remain to be worked out. As a compromise, migration can be simulated by migrating the state but leaving the actual rep data structure intact. As long as large structures are allocated separately from the main rep and can therefore be freed when the rep has "migrated", the cost should be minimal.

- The complexity of managing distributed state, even within a single object, can be high. There are additional places where deadlocks and races conditions can occur that do not appear with a single shared object.

  Experience with Tornado, and its predecessor Hurricane [Unrau *et al.*, 1995], suggests that a few key design patterns [Gamma *et al.*, 1995] are sufficient to capture the common uses of distributed data structures. Clearly more work is necessary to capture these patterns in reusable classes to make them more accessible to systems programmers in general.

8.3 Clustered Object Garbage Collection

Tornado provides a semi-automatic garbage collection scheme for clustered objects that allows clustered object references to be safely used at any time without fear of the object being destroyed while being used.

Benefits.

- Because the object state is guaranteed to persist as long as references to the object exist, many of the locking issues that relate to races with the object's destruction are eliminated.

- Since it is always safe to use clustered object references independent of which locks are currently held, all object locking can be hidden from its clients and encapsulated within the object, thereby increasing modularity in the overall design.

- By eliminating the need for a lock to protect against destruction, locking read-only objects is no longer necessary. Similarly, lock-free approaches are safe to use.

Open Issues.

- Because garbage collection relies on the count of active processes in the kernel (or server) to return to zero periodically, it is possible for the garbage collection system to be shut out for unbounded periods of time.

  In practice there is almost always some lull in requests to the kernel (or server) on a given processor, allowing garbage collection to proceed (remember that the count only has to go to zero at some point in time, no matter how short that period is). However, there are other approaches being investigated that should eliminate this concern entirely with minimal impact on performance in the common case (see Section 4.2.4).

- The garbage collection system uses a circulating token to determine when a clustered object is no longer referenced on any processor in the system. The time for the token to circulate is longer for larger systems, and may delay garbage collection for an unreasonable length of time.

  Because the only effect the delay can have is on the time to recover the memory of clustered objects, the impact is likely to be minimal, even for large systems. This is particularly true since we expect the object creation and destruction rate to be relatively low (perhaps hundreds per processor per second) compared to the rate of general-purpose memory allocation (up to tens of thousands per processor per second). In addition, with some degree of clustered object locality, the problem can be addressed through the use of multiple circulating tokens, covering different ranges of processors.

8.4 Protected Procedure Calls

The protected procedure call facility provides a highly efficiency and concurrent interprocess communication facility necessary to support servers running on multiprocessors.

Benefits.

- By localizing all data structures and eliminating locks, the PPC facility provides both low latency (competitive with the best uniprocessor IPC systems) and high concurrency (essentially unbounded).
- By ensuring client requests are always directed to the server on the local processor, it allows servers to enhance locality by allocating client-specific state local to the client.
- The PPC system provides a sufficiently general platform to also support asynchronous requests, interrupt dispatching, upcalls, and cross-processor calls, without much added complexity or performance overhead.

Open Issues.

- Because, by default, a single pool of stacks is used for all programs and the stacks are not zeroed before being assigned, there is the potential security risk of leaked information.

  There are many ways to address this problem, including the current method of simply turning off stack sharing for those programs that are concerned about potential security violations (the other obvious option for such programs is to clear their stack before returning). The most likely alternative would use a separate pool of stacks for different classes of servers. This would only add an extra word to the port entry to point to the appropriate pool and would not increase the cycle count for a PPC call.
- To keep dynamic stack allocation fast, efficient, and simple, stacks are limited to 8KB in size. This may be too small for some servers.

  So far, the stack size limit has posed no problem for the servers we have implemented. However, we expect there will be some cases where a larger stack is needed, for which a number of options were presented previously (see Section 5.3.4).

- The space required for the port table may become a problem in larger systems. This is because the total space required for all port tables (assuming a simple linear table) grows as the number of processors multiplied by the total number of programs that can be run simultaneously.

  In order to support larger systems, we expect we will need to switch from the single linear table to a partitioned system where widely accessed servers that span the entire system are placed in a smaller linear table and the regular user programs are placed in a hash table. This should have minimal impact on performance for system servers, as it only requires the addition of a simple range check on the port, and acceptable performance for calls to regular user programs that require a hash table lookup.

- The infrastructure to handle outstanding PPC calls to migrating processes has not yet been implemented.

  Although migration adds complications to the implementation, it requires no changes to the basic underlying design or architecture, and hence we feel confident it can be added with minimal impact on performance or overall design complexity.

8.5 Locks

The Tornado locking subsystem provides both time and space efficient locks in conjunction with an infrastructure that supports a variety of lock debugging functions.

Benefits.

- Tornado locks require only one or two bits of permanent state, depending on whether the lock is a spin-only or spin-then-block lock.

- On modern super-scalar processors, the Tornado locks are as efficient as standard test-and-set locks.

Open Issues.

- As Tornado is scaled up to larger systems, it may become necessary to include spinning queue-based locks to assist with scalability.

  As discussed earlier in Section 6.1.4, a number of strategies based on three-phase locks should be able to address this issue without hurting the common case time or space costs.

- As with all spin-then-block locks, choosing the spin time can be difficult. This is further complicated by false sharing cache miss effects.

  This is a general open issue, but one which can likely be addressed with the use of cache-miss counters and cycle counters commonly available on current microprocessors. Further research into this direction would appear warranted.

8.6 Memory Allocation

The Tornado memory allocator addresses within a single framework the issues of speed, locality, and concurrency, as well as false sharing and NUMAness.

Benefits.

- By once again using per-processor data structures, the common cases for the allocator provide high locality and concurrency.

- As a result of the previous point, allocation and deallocation are fast; comparable to uniprocessor times.

- The design extends previous work by addressing issues of NUMA locality and false sharing, through the use of additional specialized free lists.

Open Issues.

- The NUMA and false-sharing aspects of the memory allocator are designed but not yet implemented.

  The design is complete and appears to introduce no excessive complexity to the overall architecture. We therefore have confidence that it can be implemented and provide the expected advantages described.

- The cost of multiple checks for overfull free lists and remote data may best be paid at higher levels in the memory allocator instead of at free time, reducing the frequency of such checks.

  Postponing these checks could result in a loss of locality and highly unbalanced free lists. In addition, initial results on more modern superscalar processors indicates that the current costs are completely hidden.

- There is a danger that excessive memory will become unavailable due to the number of free lists, particularly those added for NUMA support. Memory on free lists that is never allocated is essentially unavailable to other free lists.

  By choosing the size of the free lists appropriately, such lost memory can be kept to reasonable levels. In addition, periodic scans could be used to reclaim such memory if necessary. To address the problem of large number of free lists for NUMA support, it may be necessary to combine lists to more remote processors so that the total number of lists remains constant as the system grows. Despite these potential problems, the lack of existing research into multiprocessor memory allocation suggests that simply raising these issues is a necessary first step.

8.7 Overall Open Issues

- The experimental system is limited to 16 processors.

  We expect the benefits to be even more pronounced on larger systems, although it may also highlight new bottlenecks that will need to be addressed (as is to be expected for a prototype).

- Only micro-benchmarks were used, instead of real applications. Time limitations did not allow performance testing with real workloads. As a result, the true overall value of the techniques presented in this dissertation cannot be fully evaluated. Given this limitation, this work can only be treated as preliminary (as well as a motivator for future work along similar lines).

  However, given the scalability problems of commercial systems, it seems clear that the issues discussed are of critical importance to the field. Whether we have presented the right tradeoff between locality and concurrency versus sharing and global policies will require a more complete implementation and workload than that possible within the scope of this work.

Chapter 9

Lessons Learned

Tornado was built on our experience with the Hurricane operating system [Unrau *et al.*, 1995]. Hurricane was designed with scalability and NUMA multiprocessors in mind. It was built from scratch by loosely coupling smaller tightly coupled operating systems (called clusters) to form a large-scale single-system image with the performance characteristics of small-scale systems for tightly-coupled interactions, and the scalability of distributed systems.

Hurricane was a success from the perspective of achieving its goals on the hardware it was designed for. However, we found a number of problems with this architecture that drove us towards the much more radical one employed by Tornado. Since many of these problems apply to other operating systems, it is instructive to discuss a few of them, and how they are addressed in Tornado:

1. The tightly coupled Hurricane cluster has little locality, and is hence inappropriate for current systems with large caches and high cache miss latency. The object-oriented nature of Tornado and the use of clustered objects allows available per-processor locality to be exploited.

2. The high overhead of memory-based synchronization drove the Hurricane design towards complex locking protocols with relatively large-granularity locks [Unrau *et al.*, 1994]. With the low-overhead in-cache locking primitives of modern processors, the fine-grained (in object) locking strategy of Tornado has reduced complexity as well as lowered overheads and improved concurrency.

3. When a resource is widely shared, the fixed sized clusters of Hurricane resulted in high overhead due to message passing between the clusters. With clustered-objects, a single representative can be shared by any number of processors, if advantageous to do so.

4. For correctness, all components of Hurricane had to be clustered, making the implementation of every component complex. With Tornado, a new service can be centralized initially and only later distributed (without affecting the clients of the service).

5. The traditional structure of Hurricane made it difficult to explore wildly new resource management policies and implementations. With the object-oriented structure employed by Tornado, only the interfaces between objects are well defined, and it is much easier to explore policy and implementation alternatives.

The Tornado object-oriented strategy does not come without cost. Overheads include: (i) virtual function invocation, (ii) the indirection through the translation table, (iii) the space inefficiencies described in Section 3.3.5, and (iv) the intrinsic cost of modularity, where optimizations possible by having one component of the system know about the details of another are not allowed. Our experiences to-date suggest that these costs are low compared to the performance advantages of locality, and will over time grow less significant with the increasing discrepancy between processor speed and memory latency. However, we do not believe that we yet have enough experience to make strong claims.

As the adage goes "any problem in computer science can be solved by an extra level of indirection."[1] The clustered object translation table provides this level of indirection in our operating system, and we have found it useful to solve a number of problems, including:

locking protocol: The on-demand insertion of representatives into the clustered object translation table lets us track which processors have temporary references to an object, enabling the garbage-collection based locking protocol.

dynamic flexibility: Since we can (i) keep track of temporary references to an object and (ii) block out new references until all current references have completed, we have a framework for changing the class of an object without having to negotiate with all the clients of that object.

security: The translation table is used to maintain security information, making it possible for object references to be passed between address spaces. This allows Tornado clients to use a unified object-oriented methodology to requests services, whether the client is in the same address space as the service provider or not.

supplants other global tables: The translation table supplants the need for many other global tables. For example, a client accessing a file uses the reference to an object in the file server, rather than an identifier that the file system has to translate to an object reference.

Our primary goal in developing Tornado was to design a system that would achieve high performance on shared-memory multiprocessors. We believe that the performance numbers presented

[1] In a private communication, Roger Needham attributes this to David Wheeler.

in this paper illustrate that we have been successful in achieving this goal. A result that is just as important that we did not originally target was ease of development. Less experience is necessary for developers of Tornado, than, for example, Hurricane. The object-oriented strategy coupled with clustered objects makes it easier to get a correct implementation of a new service and then incrementally optimize its performance later. Also, the locking protocol has made it much easier for inexperienced programmers to develop code, both because fewer locks have to be acquired, and because objects will not disappear even if locks on them are released. This translates many deadlock and wild write failures into explicit error conditions that are easier to track down and solve.

Chapter 10

Concluding remarks

In this dissertation we examined the issues surrounding the design and implementation of an operating system for shared-memory multiprocessors. The primary issues considered were ones of locality and concurrency. By using an object-oriented structure, with each virtual and physical resource represented by an independent object, we eliminated most shared global objects from Tornado, thus reducing contention and increasing locality. To improve performance for contended components, we introduced a new structuring technique called *Clustered Objects* that allows an object to be partitioned and distributed across the machine in a manner transparent to the outside consumers of the object. As part of the clustered object system, we presented a novel semi-automatic garbage collection scheme that significantly simplifies the locking issues faced by the programmer. We also presented a novel interprocess communication facility, called the *Protected Procedure Call* facility, that provides the locality and concurrency required to allow microkernels such as Tornado to scale effectively on multiprocessors. Finally, efficient locking and memory allocation designs were presented that meet the dual requirements of an object-oriented system and a scalable shared-memory multiprocessor.

Validation of the design was realized by way of a prototype implementation of Tornado for both the NUMAchine multiprocessor and the SimOS complete machine simulator. The prototype demonstrated both the feasibility of the design as well as its performance benefits.

The contributions of this work (at the risk of repeating myself) include the following primary innovations:

- An overall system architecture that uses object-oriented design techniques to increase locality and concurrency by fully encapsulating all resources within independent objects.

- A methodology and underlying implementation that supports the distribution of object state across the processors and memory of a multiprocessor, transparently increasing locality and concurrency for the consumers of the object.

- A semi-automatic multiprocessor garbage collection system that simplifies many of the locking issues, increases modularity, and increases performance by way of reduced synchronization requirements.

- A highly efficient and concurrent interprocess communication system that achieves performance comparable to the best uniprocessor IPC system for arbitrary degrees of concurrency.

This dissertation also includes of a number of secondary innovations:

- Software locks that require only one or two bits of permanent state, and that are potentially as efficient as standard test-and-set locks.

- A dynamic memory allocation facility that addresses the issues of concurrency, NUMA locality, false sharing, and base efficiency.

- A hardware address translation system that provides new levels of flexibility through region-specific TLB miss handling, while maintaining reasonable performance through the use of run-time code specialization.

Despite the functionality present in the current incarnation of Tornado, there still remains much work to be done. Full NUMA support in the locking and memory allocation facility is required, and more components of the kernel and system servers need to be converted to take full advantage of the clustered object system. More importantly, however, is the need for a more complete evaluation of the system using more realistic workloads, in order to determine the full range of benefits and liabilities that a design such as Tornado can provide.

Work continues on all these fronts. Tornado has now been licensed to IBM so that they may continue the research within an industrial setting and with industrial resources. Concurrent development of Tornado and the successor to Tornado, Kitchawan, at IBM's T.J. Watson research center, will hopefully permit these open issues to be addressed.

Bibliography

[Abdelrahman *et al.*, 1998]
T. Abdelrahman, N. Manjikian, G. Liu, and S. Tandri. Locality enhancement for large-scale shared memory multiprocessors. *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 335–342. Springer-Verlag, Pittsburgh, PA, 1998.

[Accetta *et al.*, 1986]
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. *Proc. 1986 Summer USENIX Conference*, pages 93–112, July 1986.

[Adve and Gharachorloo, 1995]
S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report ECE Technical Report 9512, Rice University, September 1995. Also as Digital Western Research Laboratory Research Report 95/7.

[Anderson *et al.*, 1997]
J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 1–14, New York, October 1997. ACM Press.

[Anderson, 1990]
T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[Appavoo, 1998]
J. Appavoo. Clustered Objects: Initial design, implementation and evaluation. Master's thesis, University of Toronto, 1998.

[Auslander *et al.*, 1997]

M. Auslander, H. Franke, O. Krieger, B. Gamsa, and M. Stumm. Customization-lite. *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 43–48, 1997.

[Balan, 1992]

R. Balan. A scalable implementation of virtual memory HAT layer for shared memory multiprocessor machines. *USENIX Conference Proceedings*, pages 107–116, San Antonio, TX, Summer 1992. USENIX.

[Bennett, 1987]

J. K. Bennett. The design and implementation of Distributed Smalltalk. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, page 318, 1987.

[Bershad *et al.*, 1990]

B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Computer Systems*, 8(1):37–55, February 1990.

[Bershad, 1992]

B. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–212, Seattle, WA, April 27-28 1992. USENIX.

[Black *et al.*, 1991]

D. L. Black, A. Tevanian Jr. , D. B. Golub, and M. W. Young. Locking and reference counting in the mach kernel. *Proc. 1991 ICPP*, volume II, Software, pages II–167–II–173, August 1991.

[Black, 1990]

D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, 23(5):35–43, 1990.

[Bonwick, 1994]

J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference*, pages 87–98, Berkeley, CA, USA, Summer 1994. USENIX.

[Bugnion *et al.*, 1997]

E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. on Computer Systems*, 15(4):412–447, November 1997.

[Campbell *et al.*, 1991a]

M. Campbell, R. Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith, and R. Wescott. The parallelization of UNIX system V release 4.0. *USENIX Conference Proceedings*, pages 307–324, Dallas, TX, January 21-25 1991. USENIX.

[Campbell *et al.*, 1991b]

M. Campbell, R. Holt, and J. Slice. Lock granularity tuning mechanisms in SVR4/MP. *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II.)*, pages 221–228, March 1991.

[Chang and Rosenburg, 1992]

H.H.Y. Chang and B. Rosenburg. Experience porting Mach to the RP3 large-scale shared-memory multiprocessor. *Future Generation Computer Systems*, 7(2–3):259–267, April 1992.

[Chapin *et al.*, 1995a]

J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. *Proceedings of the 1995 ACM SIGMETRICS Joint International Conferentce on Measurement and Modelling of Computer Systems*, May 1995.

[Chapin *et al.*, 1995b]

J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosio, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, 1995.

[Chen and Bershad, 1993]

J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. *Proc. 14th ACM SOSP*, pages 120–133, 1993.

[Cheriton, 1984]

D. R. Cheriton. An experiment using registers for fast message-based interprocess communication. *Operating System Review*, (4):12–20, 1984.

[Cheriton, 1988]

D. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Custer, 1993]

H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[Engler *et al.*, 1995]
D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. *Proc. 15th ACM Symp. on Operating Systems Principles*, pages 251–266, 1995.

[Eykholt *et al.*, 1992]
J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, D. Stein, M. Smith, A. Shivalingiah, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing: Multithreading the System V Release 4 kernel. *USENIX Conference Proceedings*, pages 11–18, Summer 1992.

[Fenwick *et al.*, 1995]
D. M. Fenwick, D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissell. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[Fessant *et al.*, 1998]
F. Le Fessant, I. Piumarta, and M. Shapiro. An implementation of complete, asynchronous, distributed garbage collection. *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation (PLDI '98)*, pages 152–161, 1998.

[Ford and Lepreau, 1994]
B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. *Proc. USENIX Technical Conference*, pages 97–114, 1994.

[Gamma *et al.*, 1995]
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[Gamsa *et al.*, 1994]
B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. *Proceedings of the 1994 International Parallel Processing Symposium*, pages 208–211, Boca Raton, FL, August 1994.

[Gharachorloo *et al.*, 1990]
K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, page 15, June 1990.

[Greenwald and Cheriton, 1996]
M. Greenwald and D.R. Cheriton. The synergy between non-blocking synchronization and operating system structure. *Symp. on Operating System Design and Implementation*, pages 123–136, 1996.

[Grunwald *et al.*, 1993]
D. Grunwald, B. G. Zorn, and R. Henderson. Improving the cache locality of memory allocation. *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, pages 177–186, 1993.

[Gupta *et al.*, 1991]
A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel application. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, 1991.

[Hamilton and Kougiouris, 1993]
G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. *Proc. of the 1993 Summer Usenix Conference*, 1993.

[Herlihy and Moss, 1991]
M. Herlihy and J. Moss. Lock-free garbage collection for multiprocessors. *Proceedings of the Third Symposium on Parallel Algorithms and Architectures*, pages 229–236, July 1991.

[Herlihy, 1993]
M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, November 1993.

[Hill and Larus, 1990]
M. D. Hill and J. R. Larus. Cache considerations for multiprocessor programmers. *CACM*, 33(8):97–102, August 1990.

[Hjalmtysson and Gray, 1998]
G. Hjalmtysson and R. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. *USENIX 1998 Annual Technical Conference*. USENIX, 1998.

[Homburg *et al.*, 1995]
P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and Wi. de Jonge. An object model for flexible distributed systems. *Proc. of the 1st Annual ASCI Conference*, pages 69–78, 1995.

[Hutchinson and Peterson, 1991]
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.

[Jacob and Mudge, 1998]
B. L. Jacob and T. N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. *Proc. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-8)*, 1998.

[Jeremiassen and Eggers, 1995]
T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 179–188, July 1995.

[Kagi *et al.*, 1995]
A. Kagi, N. Aboulenein, D. C. Burger, and J. R. Goodman. An analysis of the interactions of overhead-redudincg techniques for shared-memory multiprocessors. *Proc. International Conference on Supercomputing*, pages 11–20, July 1995.

[Karlin *et al.*, 1991]
A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[Kontothanassis *et al.*, 1997]
L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[Krieger *et al.*, 1994]
O. Krieger, M. Stumm, and R. Unrau. The Alloc Stream Facility: A redesign of application-level stream I/O. *IEEE Computer*, 27(3):75–82, March 1994.

[Krieger, 1994]
O. Krieger. *HFS: A flexible file system for shared memory multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, 1994.

[Liedtke *et al.*, 1997]
J. Liedtke, K. Elphinstone, S. Schonberg, H. Hartig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance. *The 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[Liedtke, 1993]
J. Liedtke. Improving IPC by kernel design. *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 175–188, North Carolina, December 1993.

[Lim and Agarwal, 1994]
B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI),*, pages 25–35, October 1994.

[LoVerso *et al.*, 1991]
S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 UNIX filesystem (UFS). *USENIX Conference Proceedings*, pages 207–218, Dallas, TX, January 21-25 1991. USENIX.

[Magnussen *et al.*, 1994]
P. Magnussen, A. L., and Erik Hagersten. Queue locks on cache coherent multiprocessors. *8th IPPS*, pages 26–29, 1994.

[Makpangou *et al.*, 1994]
M. Makpangou, Y. Gourhant, J.P. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, July 1994.

[Massalin and Pu, 1991]
H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, University of Columbia, 1991.

[McCrocklin, 1995]
Drew McCrocklin. Scaling Solaris for enterprise computing. *Spring 1995 Cray Users Group Meeting*, 1995.

[McKenney and Slingwine, 1993]
P. E. McKenney and J. Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. *USENIX Technical Conference Proceedings*, pages 295–305, San Diego, CA, Winter 1993. USENIX.

[McKusick and Karels, 1988]
M. K. McKusick and M. J. Karels. Design of a general purpose memory allocator for the 4.3BSD UNIX kernel. *USENIX Conference Proceedings*, pages 295–303, San Francisco, Summer 1988. USENIX.

[Mellor-Crummey and Scott, 1991]

J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[Mitchell *et al.*, 1994]

J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P.Kougiouris, P. Madany, M. Nelson, M. Powell, and S. Radia. An overview of the Spring system. *Proceedings of Compcon Spring 1994*, February 1994.

[Ousterhout *et al.*, 1988]

J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23, February 1988.

[Paciorek *et al.*, 1991]

N. Paciorek, S. Lo Verso, and A. Langerman. Debugging multiprocessor operating system kernels. *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II)*, pages 185–202. USENIX, Atlanta GA, March 21 - 22 1991.

[Peacock *et al.*, 1992]

J. K. Peacock, S. Saxena, D. Thomas, F. Yang, and W. Yu. Experiences from multithreading System V Release 4. *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 77–92, March 1992.

[Pike *et al.*, 1991]

R. Pike, D. Presotto, K. Thompson, and G. Holzmann. Process sleep and wakeup on a shared-memory multiprocessor. *Proc. Spring EurOpen Conf.*, pages 161–166, 1991.

[Popek and Walker, 1985]

G. Popek and B. Walker. *The LOCUS Distributed System Architecture*. The MIT Press, Cambridge, Mass, 1985.

[Presotto, 1990]

D. L. Presotto. Multiprocessor streams for Plan 9. *Proc. Summer UKUUG Conf.*, pages 11–19, London, July 1990.

[Ritchie and Thompson, 1974]

D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *CACM*, 17(7):365–375, July 1974.

[Rosenblum *et al.*, 1995]

M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. *The 15th ACM Symposium on Operating Systems Principles*, December 1995.

[Rosenblum *et al.*, 1997]

M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[Ruane, 1990]

L. M. Ruane. Process synchronization in the UTS kernel. *Computing Systems*, volume 3,3, pages 387–422. USENIX, Summer 1990.

[Savage *et al.*, 1997]

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 27–37, New York, October 1997. ACM Press.

[Saxena *et al.*, 1993]

S. Saxena, J. K. Peacock, F. Yang, V. Verma, and M. Krishnan. Pitfalls in multithreading SVR4 STREAMS and other weightless processes. *USENIX Technical Conference Proceedings*, pages 85–96, San Diego, CA, Winter 1993. USENIX.

[Scales *et al.*, 1996]

D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, pages 174–185, October 1996.

[Sciver, 1990]

J. Van Sciver. Zone garbage collection. In USENIX Association, editor, *Workshop proceedings: Mach, October 4–5, 1990, Burlington, Vermont*, pages 1–16, Berkeley, CA, USA, October 1990. USENIX.

[Talbot, 1995]

Jacques Talbot. Turning the AIX operating system into an MP-capable OS. *USENIX 1995 Technical Conference Proceedings*, January 1995.

[Thacker and Stewart, 1987]

C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. *Proc. 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–172, 1987.

[Torrellas *et al.*, 1992]

J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 162–174, September 1992.

[Torrellas *et al.*, 1994]

J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Transactions on Computers*, 43(6), June 1994.

[Unrau *et al.*, 1994]

R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. *Proc. 1st Symp. on Operating Systems Design and Implementation (OSDI)*, November 1994.

[Unrau *et al.*, 1995]

R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[Vahalia, 1996]

U. Vahalia. *UNIX Internals*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[Valois, 1995a]

J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[Valois, 1995b]

J. D. Valois. Lock-free linked lists using compare-and-swap. *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, 2–23 1995.

[Vranesic *et al.*, 1991]

Z. G. Vranesic, M. Stumm, R. White, and D. Lewis. The Hector multiprocessor. *IEEE Computer*, 24(1), January 1991.

[Vranesic *et al.*, 1995]

Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, April 1995.

[Wilk, 1997]

D. Wilk. Hierarchical application-oriented physical memory management. Master's thesis, University of Toronto, 1997.

[Wilson *et al.*, 1995]

P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[Wilson, 1992]

P. R. Wilson. Uniprocessor garbage collection techniques. *Intl. Workshop on Memory Management*. Springer-Verlag, 1992.

[Xia and Torrellas, 1996]

C. Xia and J. Torrellas. Improving the data cache performance of multiprocessor operating systems. *HPCA-2*, 1996.

[Zajcew *et al.*, 1993]

R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An OSF/1 UNIX for massively parallel multicomputers. *USENIX Technical Conference Proceedings*, pages 449–468, San Diego, CA, Winter 1993. USENIX.