Clustered Objects

by

Jonathan Appavoo, B.Sc., M.Sc.

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

Clustered Objects

Jonathan Appavoo, B.Sc., M.Sc.
for the degree of Doctor of Philosophy
Department of Computer Science
University of Toronto, 2005

Supervisor: Michael Stumm

In this dissertation we establish that the use of distribution in the implementation of a shared memory multi-processor operating system is both feasible and able to substantially improve performance of core operating system services. Specifically we apply distribution in the form of replication and partitioning in the construction of K42, a shared memory multi-processor operating system. *Clustered Objects*, a software construction for the systematic and selective application of distribution to objects of K42's object oriented system layer, is presented. A study of the virtual memory services of K42 is conducted in which distribution is applied to key virtual memory objects to enable performance improvements. The distributed versions of these objects substantially improve scalability, and specifically improve throughput of a standard multiuser benchmark by 68% on a 24 way multi-processor. Additionally, a methodology for the dynamic hot-swapping of *Clustered Object* instances is presented as a means for enabling dynamic adaptation. Motivated by the desire to hot-swap between centralized and distributed implementations of *Clustered Objects*, the methodology presented is correct, efficient and integrated with the *Clustered Object* support.

Contents

Glossary

COS: Clustered Object System: the name of the Clustered Object infrastructure in K42.

COSMgr: Clustered Object System Manager: a distributed Clustered Object which implements the core features of the COS.

FCM: File Cache Manager: the object which manages the resident pages for an individual file.

FR: File Representative: the object which represents an open file in the kernel and is used to initiate IO operations to a file system.

Global PM: Global Page Manager: the object which manages the physical page frames across the entire system.

HAT: Hardware Address Translation Object: the object used to represent and manipulate hardware page tables. K42 utilizes a per-processor, per-process page tables.

miss-handling: the process initiated when a clustered object is accessed on a processor on which there is no local translation for the object. The translation mechanism invokes a method of the Root of the clustered object to carry out the object's miss-handling behavior. Miss-handling is leveraged for lazy Representative creation and as a general mechanism for redirecting calls on a clustered object.

PM: Page Manager: the object which manages the physical page frames allocated to an individual Process.

Process Object: the object which represents a running process.

RCU: Read-Copy-Update: a synchronization discipline which utilizes quiescent states to safely partition changes to a data structure.

Region: the object which represents a mapping of a portion of a processes address space to a portion of a file.

Representative: the object which serves as the access point for a clustered object instance. Single or multiple Representatives can be associated with a single clustered object instance.

Root: the internal central object which serves as the anchor for a clustered object instance. All Representatives have a pointer to the root of the Clustered Object instance to which they belong.

VMM: Virtual Memory Management: a core service of any modern operating system. Manages mapping of virtual addresses to physical page frames which contain the data to be accessed.

Chapter 1

Introduction

1.1 Problem Statement

Despite decades of research and development into Shared Memory Multi-Processor (*SMP*) operating systems, achieving good scalability for general purpose workloads, across a wide range of processors, has remained elusive. "Sharing" lies at the heart of the problem. By its very nature, sharing introduces barriers to scalability and comes from three main sources. First, sharing can be intrinsic to the workload. For example, a workload may utilize a single shared file to log all activity of multiple processes, or a multi-threaded application may use a shared data array across multiple processors. Secondly, sharing also arises from the data structures and algorithms employed in the design and implementation of the operating system (OS). For example, an operating system may be designed to manage all of the physical memory as a single shared pool and implement this management using shared data structures. Thirdly, sharing can occur in the physical design and protocols utilized by the hardware. For example, a system utilizing a single memory bus and snooping based cache coherence protocol requires shared bus arbitration and global communications for memory access, even for data located in independent memory modules.

General purpose operating systems provide a basic framework for efficiently utilizing computer hardware. To facilitate computer use, the operating system provides an abstract model of a computer system through a set of common interfaces realized by operating system services. A critical issue in the development of operating system services is to enable efficient application utilization of hardware resources. Operating systems for multi-processors must ensure correctness and scalability of the system services in order to allow the parallelism afforded by the hardware to be exploited for both general purpose and explicit parallel workloads. A thorough overview of operating systems

and basic concurrency can be found in a number of undergraduate-level textbooks on this subject matter [113, 121, 127, 130, 135].

Operating systems are unique, from a software perspective, in their requirement to support and enable parallelism rather than exploiting it to improve their own performance. An operating system must ensure good overall system utilization and high degrees of parallelism for those applications which demand it. To do this, operating systems must: 1) utilize the characteristics of the hardware to exploit parallelism in general purpose workloads, and 2) facilitate concurrent applications, which includes providing models and facilities for applications to exploit the parallelism available in the hardware.

It is critical that an OS reflect the parallelism of the workloads and individual applications to ensure that the OS facilities do not hinder overall system or individual application performance. This point is often overlooked. Smith alludes to the requirements of individual applications, noting that the tension between protection and performance is particularly salient and difficult in a parallel system and that the parallelism in one protection domain must be reflected in another [123]. In other words, to ensure that a parallel application within one protection domain can realize its potential performance, all services in the system domains that the concurrent application depends on must be provided in an equally parallel fashion. It is worth noting that this is true not only for individual applications but also for all applications forming the current workload on a parallel system: the demands of all concurrently executing applications must be satisfied with equal parallelism in order to ensure good overall system performance.

Achieving scalable performance requires minimizing all forms of sharing, which is also referred to as maximizing locality. On shared memory multi-processors (SMPs), locality refers to the degree to which locks are contended and data — including the locks themselves — are shared amongst threads running on different processors. The less contention on locks and the less data is shared, the higher the locality. Maximizing locality on SMPs is critical, because even minute amounts of (possibly false) sharing can have a profound negative impact on performance and scalability [19,49].

The work of this dissertation is focused on the application of techniques to the construction of systems software to improve locality. We assert that, with considerable effort, we can reduce sharing in the operating system in the common case and achieve good scalability on a wide range of processors and workloads, where performance is limited by the hardware and inherent scalability of the workload. The most obvious source of sharing that is within the control of the operating system designer is the data structures and algorithms employed by the operating system. However,

Gamsa observes that, prior to addressing specific data structures and algorithms of the operating system in the small, a more fundamental restructuring of the operating system can reduce the impact of sharing induced by the workload by minimizing sharing in the operating system structure [49]. Gamsa's work uses an object-oriented model to create individual, independent instances of operating system resources which can be instantiated as necessary to meet many of the parallel demands of the workload. This allows accesses to independent resources to be directly reflected to independent software structures, thus avoiding unnecessary sharing due to shared meta structures.

The above approach alone, however, does not eliminate sharing, but rather helps limit it to the paths and components which are shared due to the workload. For example, consider the performance of a concurrent multi-user workload on K42, a multi-processor operating system constructed using Gamsa's design. Assume a workload that simulates multiple users issuing unique streams of standard UNIX commands with one stream issued per processor. Such a workload, from a user's perspective, has a high degree of independence and little intrinsic sharing. Despite the fact that K42 is structured in an object-oriented manner, with each independent resource represented by an independent object instance, at four processors, throughput is 3.3 times that of one processor and at 24 processors, throughput is 12.5 times that of one processor. Ideally, the throughput would increase linearly with the number of processors. Closer inspection reveals that the workload induces sharing on OS resources, thus limiting scalability. We contend that in order to ensure the remaining sharing does not limit our performance, we must use distribution, partitioning and replication to remove sharing in the common paths. Using distributed implementations for key virtual memory objects, and running the same workload as above, the OS yields a 3.9 times throughput at four processors and a 21.1 times throughput at 24 processors. (See Chapter 7 and Figure 7.1 for details of these results.) The work presented in this dissertation describes techniques and methods for building such data structures and applying them to the operating system in order to achieve scalable performance for standard workloads.

The development of high-performance, parallel systems software is non-trivial. The concurrency and locality management needed for good performance can add considerable complexity. Fine-grain locking in traditional systems results in complex and subtle locking protocols. Adding per-processor data structures in traditional systems leads to obscure code paths that index per-processor data structures in ad-hoc manners. *Clustered Objects* were developed as a model of *partitioned objects* to simplify the task of designing high-performance SMP systems software [144]. In the partitioned object model, an externally visible object is internally composed of a set of distributed

Representative objects. Each Representative object locally services requests, possibly collaborating with one or more other Representatives of the same Clustered Object. Cooperatively, all the Representatives of the Clustered Object implement the complete functionality of the Clustered Object. To the clients of the Clustered Object, the Clustered Object appears and behaves like a traditional object. The distributed nature of Clustered Objects make them ideally suited for the design of multi-processor system software, which often requires a high degree of modularity and yet benefits from the sharing, replicating and partitioning of data on a per-resource (object) basis. Clustered Objects are conceptually similar to design patterns such as facade [47]; however, they have been carefully constructed to avoid any shared front end, and are primarily used for achieving data distribution. Some distributed systems have explored similar object models. Specifically, Clustered Objects are similar to other partitioned object models, such as Fragmented Objects [17, 83] and Distributed Shared Objects [62, 136], although the latter have focused on the requirements of (loosely coupled) distributed environments. In contrast, Clustered Objects are designed for (tightly coupled) shared memory systems. Section 2.2 discusses related work in more detail.

Our use of the word *distributed* throughout refers to the division of data across a shared memory multi-processor complex. Distribution does not imply message passing, but rather, *distribution* across multiple memory locations all of which can be accessed via hardware supported shared memory. We *distribute* data across multiple memory locations in order to: (*i*) optimize cache line access, (*ii*) increase concurrency, and (*iii*) exploit local memory on architectures which have non-uniform memory access, where some memory modules may be cheaper to access from a given processor.

A key to achieving high performance on a multi-processor is to use per-processor data structures whenever possible, so as to minimize inter-processor coordination and shared memory access. In this dissertation per-processor data structures refers to the use of a separate instance of a data structure for each processor. The software is constructed, in the common case to access and manipulate the instance of the data structure associated with the processor on which the software is executing. The use of per-processor data structures is to improve performance by enabling distribution, replication and partitioning of stored data. In general, access to of any of the data structure instances by any processor is not precluded given the shared memory architectures we are targeting. In contrast, the ability to access all data structure instances via shared memory is often used to implement coordination and scatter-gather operations as necessary.

Typically there are two performance characteristics that are used when evaluating parallel pro-

gram performance: 1) absolute performance, and 2) performance improvement due to parallelism (speedup or associated scalability measures) [37]. In this dissertation we will focus on the latter, but to ensure that we are not seeing improved scalability at the cost of unreasonable base uniprocessor performance, we present the results from a traditional Operating System (Linux, in particular) for comparison for the full system workloads. In general, when we consider performance, we are interested in evaluating the operating system's ability to satisfy concurrent user-level service requests, associated with a given user-level workload running on some number of processors. To do this we evaluate performance by running a workload on a range of processor configurations. As we increase the number of processors, we scale the user-level workload, and measure performance as either throughput or execution time. From this data, we plot either throughput, speedup or slowdown. Note that, although all of our implementation work is limited to the operating system, we do not explicitly measure performance of the operating system, but only measure application performance, which indirectly includes OS overheads. We make no modifications to applications, hence we can observe the impact of OS effects.

In this dissertation we describe the design and implementation of distributed data structures in the construction of a shared memory multi-processor operating system. We standardize Clustered Objects via a set of protocols which provide a distributed object-oriented model, permitting incremental development while fully leveraging hardware-supported shared memory. We present the application of distribution, via distributed Clustered Objects we implemented, to optimize a critical systems service, namely the memory management subsystem. Distributed implementations can offer better scalability. However, distributed implementations typically optimize certain operations, improving their scalability, but often increase costs of other operations at the same time. They also typically suffer greater overheads when scalability is not required. In order to provide a means of coping with the tradeoffs of using distributed implementations, we also present a technique we developed for dynamically replacing a live Clustered Object instance with a newly created compatible instance. This mechanism can be used to switch between non-distributed and distributed implementations dynamically at run time and additionally enable other forms of dynamic adaptation.

1.2 Motivation

We contend that MP OS scalability is important and is becoming increasingly relevant. An increasingly large number of commercial 64-way or even 128-way MPs are being used as Web or Data Base (DB) servers. Next generation game stations will be 4-way SMP's, but it is not hard to envision 32- or 64-way SMP desktop PC's and game stations in the not too distant future with the expected increases in transistor density supporting relatively large-scale SMT/CMP chips. The High End Computing (HEC) community is recognizing the importance of using large-scale SMP's to overcome performance bottlenecks: the Department of Energy has expressed interest in using K42 as their core operating system for their HEC needs, and K42 is being used in one of the systems being built for DARPA's High Productivity Computing Systems Program. While our efforts have focused on SMP operating system software, the techniques we describe are applicable to any transaction-driven service (i.e. OS, Web server, DB, etc.) where parallelism is not generated within the service, but is a result of parallel requests by the workload using the service. The importance of scalability can also be witnessed by the large efforts that have gone into parallelizing Linux and commercial operating systems, such as AIX, SunOS, and Irix over the last several years[1].

1.3 Research Context: Tornado and K42 Operating Systems

The work presented in this dissertation has been conducted in the context of two operating systems projects, Tornado and K42. The Tornado operating system was designed and implemented by the systems research group at the University of Toronto, primarily focusing on the exploration of techniques for supporting large Non-Uniform Memory Access (NUMA) multi-processors. Research into the Tornado operating system ended in 1999. The K42 operating system project began in 1998 at IBM. K42 is based on Tornado and the University of Toronto research group has collaborated with the IBM research group in K42's development. Specifically, the University of Toronto research group has been working on the K42 source base since 1999 and has focused on the scalability aspects of K42. The majority of the implementation and experimentation we have conducted and documented in this thesis has been done in K42. The remainder of this section gives a brief overview of K42. Over the course of the work covered in this dissertation I have been working as part of both the University of Toronto research group and the IBM research group. In 2003 I joined the

[1] In industry, large efforts have been spent on improving scalability of operating systems, albeit with disappointing results. Unfortunately very little of this work has been published.

IBM group in a full time capacity and continue to work on K42.

In this dissertation we will explicitly disambiguate the work done by the larger University of Toronto group, the work done by the larger IBM research group and my individual work. When referring to my work, as supervised by Professor Stumm, I will use the terms, 'I', 'we' and 'our'. The attribution of work done by the larger groups of the University of Toronto and IBM will be made explicitly.

Providing a well-structured kernel is a primary goal of the K42 project, but performance is the central concern. Some research operating system projects have taken particular philosophies and have followed them rigorously in order to fully examine their implications. While the IBM research group follows a set of design philosophies in K42, compromises are made for the sake of performance. The principles that guide the design include: 1) structuring the system using modular, object-oriented code, 2) designing the system to scale to very large shared-memory multi-processors, 3) avoiding centralized code paths and global data structures and locks, 4) leveraging performance advantages of 64-bit processors, 5) moving as much system functionality as possible to application libraries, and 6) moving system functionality from the kernel to server processes. The first four are direct consequences of the Tornado [48] work and the contributions of my dissertation.

Two of the goals of the K42 project are:

- **Performance:** A) Scale up to run well on large multi-processors and support large-scale applications efficiently. B) Scale down to run well on small multi-processors. C) Support small-scale applications as efficiently on large multi-processors as on small multi-processors.

- **Adaptability:** A) Allow applications to determine (by choosing from existing components or by writing new ones) how the operating system manages their resources. B) Autonomically have the system adapt to changing workload characteristics.

K42 is structured around a client-server model (see Figure 1.1). The kernel is one of the core servers, currently providing memory management, process management, IPC infrastructure, base scheduling, networking and device support. In the future the IBM research group plan to move networking and device support into user-mode servers.

The layer above the kernel contains applications and system servers, including the NFS file server, name server, socket server, pseudo terminal (pty) server, and pipe server. For flexibility, and to avoid IPC overhead, as much functionality as possible is implemented in application-level libraries. For example, all user-level thread scheduling is done by a user-level scheduler linked into

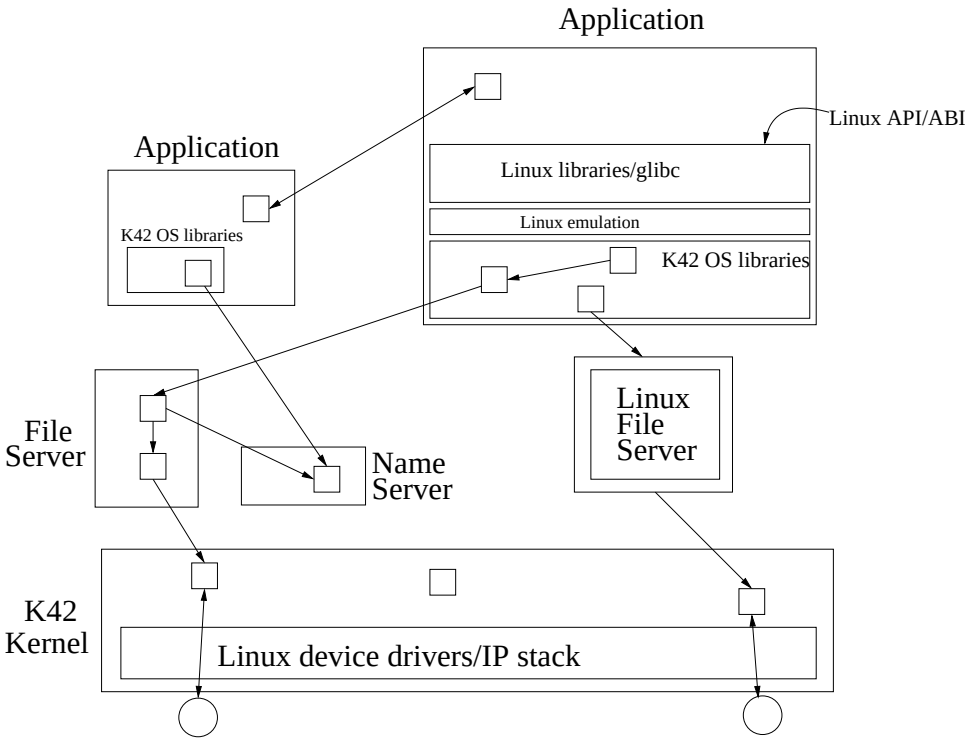


Figure 1.1: Structural Overview of K42

each process.

All layers of K42, the kernel, system servers, and user-level libraries, make extensive use of object-oriented technology. All inter-process communication (IPC) is between objects in the client and server address spaces. A *stub compiler* is used to interpret custom syntax decorations on the C++ class declarations to automatically generate IPC calls from a client to a server, and these IPC paths have been optimized to have good performance. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls.

From an application's perspective, K42 supports the Linux API and Linux ABI. This is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects. When writing an application to run on K42, it is possible to program to the Linux API or directly to the native K42 interfaces. All applications, including servers, are free to reach past the Linux interfaces and call the K42 interfaces directly. Programming against the native interfaces allows the application to take advantage of K42 optimizations. The translation of standard Linux system calls is done by intercepting system calls and implementing them with K42 code.

1.4 Contributions

Designing and implementing K42 as a fully functioning operating system has been a large effort involving a number of people. This section identifies my individual contributions.

K42 is a descendant of the Tornado operating system designed and implemented at the University of Toronto. Tornado was based on a set of synergistic structuring principles and OS mechanisms for the construction of scalable multi-processor operating systems [49]. Gamsa et al. presented some initial micro-benchmark performance results for Tornado (as illustrated in Figure 3.2), which primarily leveraged an object-oriented decomposition and manually applied distribution to its infrastructure [48, 49]. Tornado made very limited use of distribution with respect to its individual objects and higher level services.

It was concluded that better performance and scalability was achieved by utilizing an object-oriented structure, with independent objects representing individual resources. Such a structure allows independent service requests to be serviced by independent objects, thus reducing sharing within the OS. Recognizing that some objects will inherently be accessed in a shared fashion, Gamsa implemented a set of low level mechanisms which he felt would facilitate constructions of distributed implementations. My work began as a study exploring the use of these mechanisms to construct distributed implementations of some example objects [3]. The bottom two layers of Figure 1.2 isolate Gamsa's contributions to the Clustered Object research. Fundamentally, only a single distributed Clustered Object implementation was used in Tornado, which we developed based on my initial study. The following mechanisms were developed by Gamsa with a particular focus on the base scalability of these mechanisms, ensuring that no shared memory or global synchronization is required [48]:

- The use of per-processor indirection tables, mapped to the same virtual address but backed by independent physical memory, to enable efficient translation of a global Clustered Object identifier to a processor-specific Representative without requiring remote memory access in the common case.

- Manipulation of C++ virtual method dispatch to enable lazy establishment of the per-processor translations of a Clustered Object, enabling both per-object lazy instantiation and ensuring that the overhead in the common case is a pointer dereference and a virtual method dispatch, without requiring remote memory accesses.

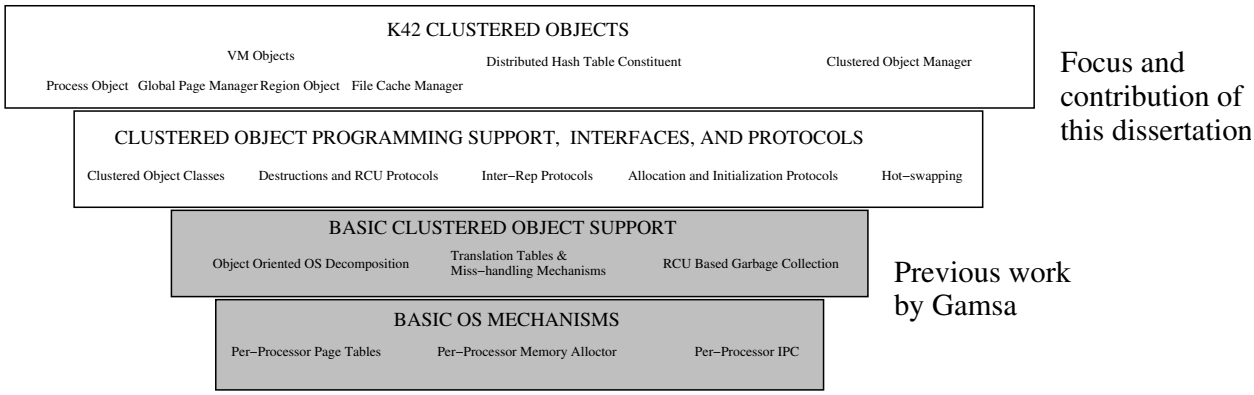


Figure 1.2: Clustered Object Research Map

- The generation of auxiliary objects and infrastructure which implement a security model and utilize an efficient, interprocess protected procedure call mechanism [51] to expose the services of a Clustered Object located in one address space, to threads executing in another, in a scalable and secure manner.

My dissertation focuses on the implementation of Clustered Objects, including software support for their development in the form of a set of base classes that formalize the internal structure of a Clustered Object and facilitate the construction of both distributed and centralized implementations. We specifically study the use of distributed implementations to optimize the Virtual Memory system of K42.

In order to address the needs of large SMP's, K42 adopted Tornado's fundamental structure. We continued our work on Clustered Objects on K42, as its initial code base was being developed, re-implementing some of the lower level Clustered Object mechanisms. Of the lower two levels depicted in Figure 1.2, we specifically implemented from scratch new versions of the Translation Tables, Miss-handling Mechanisms, and RCU-Based Garbage Collection for K42. We then implemented an interface to these basic Clustered Object services in terms of a Clustered Object System (COS) manager and a set of base C++ classes. This software supported a design of Clustered Objects derived from my initial study of Clustered Objects in Tornado, supporting both centralized (shared) and distributed implementations. We then adapted all existing K42 system objects into centralized Clustered Objects[2] so they could easily be migrated to distributed implementations later. From that point on, all system objects constructed by any developer of K42 were Clustered Objects utilizing my infrastructure. It was critical to my work to have established the ubiquitous use of

[2]A centralized or shared Clustered Object does not utilize distribution, but rather uses one Representative for all processors and as such is functionally similar to a standard C++ object.

Clustered Objects for all K42 objects. It has allowed me to consider the construction of a compatible distributed implementation for any existing object.

We then refined the basic Clustered Object mechanisms and developed more advanced base classes for Clustered Object development. These base classes codify a set of protocols for destruction, inter-rep communication (the co-ordination of the Representatives of a single Clustered Object instance), lazy allocation, initialization, and hot-swapping (the dynamic replacement of one Clustered Object instance with another). As illustrated in Figure 1.2, this portion of my work (third layer from the bottom) is the next level of Clustered Object support, which permits the development of compatible shared and distributed Clustered Object implementations on top of the basic mechanisms.

The next component of my work entailed the development of distributed Clustered Object implementations of key K42 objects. We specifically focused on distributed implementations of the Virtual Memory Management (VMM) Objects of K42, in order to optimize the performance of SDET, a general purpose operating system workload [125]. We explored various techniques for incrementally developing compatible distributed implementations of the following K42 VMM objects: Process, Page Managers, Region and File Cache Managers. Of these, I have individually developed all but the Page Managers[3] from scratch. We investigated their performance characteristics and tuned them with respect to SDET.

The hot-swapping of Clustered Object instances is ongoing research which began as a study of how to construct the mechanisms to support the dynamic replacement of an instantiated Clustered Object with another at run-time [5]. My main contribution was the development of the Clustered Object infrastructure used to implement the hot-swapping protocol, along with jointly designing the protocol itself. Hui [63], implemented a specialized Clustered Object which utilizes the Clustered Object infrastructure to implement the hot-swapping protocol. Since then we have continued to evolve the hot-swapping support and protocol. The hot-swapping work is still at its early stages and to date we have a set of mechanisms but have not explored the design and implementation of policies which utilize the mechanisms in order to adapt the system in a more autonomic fashion.

Although Clustered Objects do encapsulate and hide distribution of an implementation from the clients of an object, they do not alleviate the complexity in the internal implementation (even though they localize the concern). Developing a distributed Clustered Object has all the challenges

[3]Another member of the IBM K42 team developed the first distributed implementation of the Page Managers in order to address a measured performance problem.

of building a tightly coupled distributed application. In 1998, we presented an initial evaluation of Clustered Objects and their internal design [3]. One of the results of that work was an internal model for Clustered Objects which formalized the internal components and their relationships. That model was used as the basis for a new Clustered Object implementation in K42.

In summary, in this work we have:

1. developed infrastructure and protocols necessary to support the development of Clustered Objects. When designing such support, we guided decisions based on performance and generality, with the goal of providing an infrastructure which can be applied and reused by others when developing Clustered Objects.

2. applied distributed data structures and algorithms at the granularity of individual system objects in order to improve performance of a core system service, namely virtual memory management.

3. enabled the dynamic replacement of Clustered Objects to permit online adaptation in order to be able to adapt to changing workloads.

One might question the utility of spending great effort on distributed data structures if prohibitive sharing exists in the applications. However, distributed data structures are required even when the user applications are not parallel in nature. In subsequent chapters, we show that in order to obtain good scalability for standard multi-user workloads, fundamental operating system data structures must be partitioned and distributed. Even if the user applications are not parallel in nature, the operating system itself can critically limit overall scalable performance.

1.5 Outline of this Dissertation

The next chapter reviews related research. Chapter 3 highlights the motivation for this work and provides an overview of the Clustered Object research. Chapter 4 describes application of distribution to the construction and optimization of K42, focusing on a case study of the virtual memory management services. Chapter 5 describes in detail the protocols and infrastructure created to support Clustered Object development. Chapter 6 discusses a technique for dynamically replacing an object on the fly. Chapter 7 presents relevant performance results. Chapter 8 concludes, discussing various observations and future work.

Chapter 2

Background and Related Work

Much of the research into multi-processor operating systems has been concerned with how to support new, different or changing requirements in OS services, specifically focusing on user level models of parallelism, resource management and hardware configuration. We will generically refer to this as support for flexibility. In contrast, the research done at the University of Toronto has pursued a performance oriented approach. The group has suggested two primary goals for the success of multi-processor operating systems:

1. Provide a structure which allows good performance and scalability to be achieved with standard tools, programming models and workloads without impacting the user or programmer. Therefore the OS must support standards while efficiently mapping any available concurrency and independence to the hardware without impacting the user level view of the system.

2. Enable high performance applications to side step standards to utilize advanced models and facilities in order to reap maximum benefits without being encumbered by traditional interfaces or policies which do not scale.

Surveying the group's work over the last several years, two general requirements for high performance systems software have been identified:

1. *Reflect the Hardware:*

    - Match scalability of the hardware in the systems software. Ensure software representations, management and access methods do not limit system scale, for example, software service centers required to manage additional hardware resources should increase with the scale of the system.

Name	*Leading Authors*	*Year*	*Institution*
Hydra [36, 79, 141, 142]	W. Wulf	1974	Carnegie Mellon University
StarOS [36],	A. K. Jones	1979	Carnegie Mellon University
Medusa [100],	J. K. Ousterhout	1980	Carnegie Mellon University
Tunis [45]	R.C. Holt	1985	University of Toronto
Dynix/PTX [13, 52, 65]	Sequent	1985	Sequent
Mach [16, 66, 107, 108, 145]	R. Rashid	1986	Carnegie Mellon University
Psyche [115–118]	M. L. Scott	1988	University of Rochester
Clouds [39, 40]	P. Dasgupta	1988	Georgia Institute of Technology
Sprite [59]	J. K. Ousterhout	1988	University of California Berkeley
Presto [15]	B.N. Bershad	1988	University of Washington
Distributed Shared Objects (DSOs) [8, 62, 136]	A. S. Tannenbaum	1988	Vrije University
Elmwood [77]	T.J. Leblanc	1989	University of Rochester
Choices [22, 23, 72]	R.H. Campbell	1989	University of Illinois
Synthesis [86–88]	H. Massalin	1989	Columbia University
Fragmented Objects (FOs) [17, 83, 120]	M . Shapiro	1989	INRIA
Mosix [10, 11]	A. Barak	1989	Hebrew University of Jerusalem
Topologies, Distributed Shared Abstractions (DSAs) [34, 35, 99, 114]	K. Schwan	1990	Georgia Institute of Technology
Concurrent Aggregates (CAs) [33]	A. A. Chien	1990	Massachusetts Institute of Technology
Paradigm/Cache Kernel [30, 31]	D. R. Cheriton	1991	Stanford University
pSather [80]	C. Lim	1993	University of California Berkeley
KTK/CTK [53, 122]	K. Schwan	1994	Georgia Institute of Technology
Hurricane [134]	M. Stumm	1994	University of Toronto
Hive [27]	J. Chapin	1997	Stanford University
Disco [20]	M. Rosenblum	1997	Stanford University
Tornado [48, 49]	B. Gamsa	1998	University of Toronto

Table 2.1: Table of main operating systems research discussed.

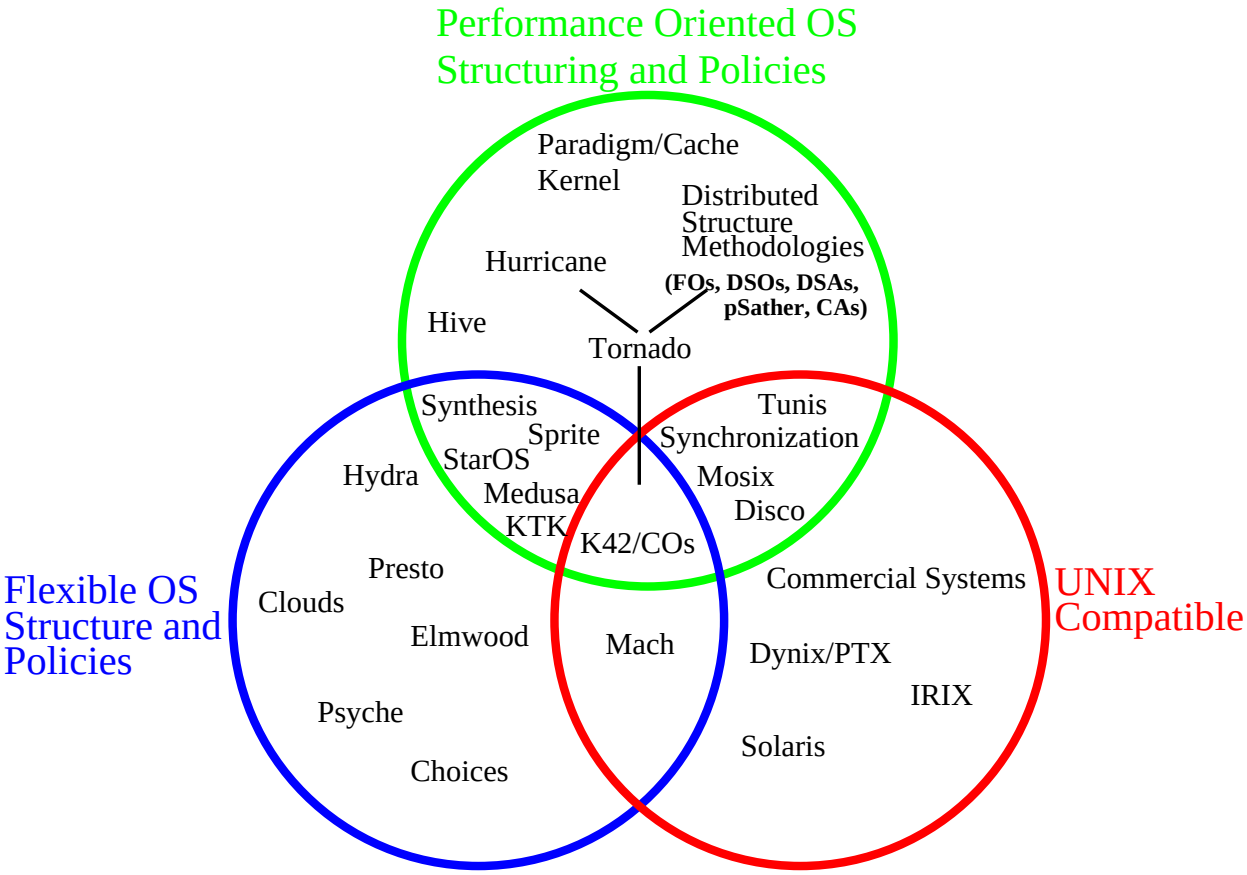


Figure 2.1: Illustration of related work with respect to K42 and Clustered Objects

- Reflect unique performance characteristics of MP hardware to maximize performance. Mirror locality attributes of the hardware in the software structures and algorithms: avoid contention on global busses and memory modules, avoid shared cache-line access and efficiently utilize replicated hardware resources.

2. Reflect the Workload:

- Map independence between applications into systems structure.
- Match the concurrency within applications in the systems structures.

In this chapter we review relevant work. We begin with a review of early MP OS research, then look at systems work directly related to the use of distributed data structures and finally conclude with a look at modern research into MP OSes. Figure 2.1 illustrates a map of the related work with respect to K42 and Clustered Objects; Table 2.1 lists the work in chronological order.

2.1 Early Multi-Processor OS Research Experience

Arguably the most complex computer systems are those with multiple processing units. The advent of multi-processor computer systems presented operating systems designers with four intertwined issues:

1. true parallelism (as opposed to just concurrency),
2. new and more complex hardware features, such as multiple caches, multi-staged interconnects and complex memory and interrupt controllers,
3. subtle and sensitive performance characteristics, and
4. the demand to facilitate user exploitation of the system's parallelism while providing standard environments and tools.

Based on the success of early multiprogramming and time sharing systems and what was viewed as fundamental limits of uniprocessor performance, early systems researchers proposed multi-processors as the obvious approach to meeting the increasing demands for general purpose computers. The designers of Multics, in 1965, went so far as to say:

> "...it is clear that systems with multiple processors and multiple memory units are needed to provide greater capacity. This is not to say that fast processor units are undesirable, but that extreme system complexity to enhance this single parameter among many appears neither wise nor economic."

Perhaps the modern obsession with uniprocessor performance for commodity systems is the strongest evidence of our inability to have successfully leveraged large scale multi-processors for general purpose computing. Large multi-processors are predominately now considered as platforms for specialized super-computing applications. We believe, however, this is changing. In an attempt to meet increasing performance demands and address memory latencies, there is renewed interest in utilizing parallel processing for commodity systems. For example, it is expected that the next generation Sony Playstation will have between 4 and 16 parallel processing units. Given this one expects future workstations and servers to adopt even more aggressive configurations.

Real world experimentation with general purpose multi-processors began as simple dual processor extensions of general purpose commercial uniprocessor hardware [7]. The general approach was to extend the uniprocessor operating system to function correctly on the evolved hardware. The

primary focus was to achieve correctness in the presence of true parallelism. This typified the major trend in industrial operating systems. They start with a standard uniprocessor system, whose programming models and environments are accepted and understood, and extend it to operate on multi-processor hardware. Various techniques for coping with the challenges of true parallelism have been explored, starting with simple techniques which ensured correctness, but yielded little or no parallelism in the operating system itself. As hardware and workloads evolved, the demand to achieve greater parallelism in the operating systems forced OS implementors to pursue techniques which would ensure correctness but also achieve higher performance in the face of parallel workload demands.

The fundamental approach taken was to apply synchronization primitives to the uniprocessor code base in order to ensure correctness. Predominantly the primitive adopted was a shared memory lock, implemented on top of the atomic primitives offered by the hardware platform. The demand for higher performance led to successively finer grain application of locks to the data structures of the operating systems. Doing so increased concurrency in the operating system at the expense of considerable complexity and loss of platform generality. The degree of locking resulted in systems whose performance was best matched to systems of a particular scale and workload demands. Despite having potentially subtle and sensitive performance profiles, the industrial systems preserved the de facto standard computing environments and achieved reasonable performance for small scale systems, which have now become widely available. It is unclear if the lack of acceptance of large scale systems is due to the lack of demand or the inability to extend the standard computing environments to achieve good performance on such systems for general purpose workloads.

In general, the industrial experience [41, 69, 71, 81, 82, 90, 102, 103, 112] can be summarized as a study into how to evolve standard uniprocessor operating systems with the introduction of synchronization primitives. Firstly, this ensures correctness and secondly, permits higher degrees of concurrency in the basic OS primitives.

In contrast to the industrial research work, the majority of the early academic research work, focused on flexibility and improved synchronization techniques. Systems which addressed flexibility include: Hydra [36, 79, 141, 142], StarOS [36], Medusa [100], Choices [22, 23, 72], Elmwood [77], Presto [15], Psyche [115–118], Clouds [39, 40]. With the exception of Hydra, StarOS and Medusa, very few systems actually addressed unique multi-processor issues or acknowledged specific multi-processor implications on performance. In the remainder of this section we highlight work which either resulted in relevant performance observations or attempted to account for multi-processor

performance implications in operating system construction.

In 1985 the Tunis [45] operating system, created by another group at the University of Toronto under the direction of Professor R. C. Holt, was one of the first systems to focus on the importance of locality rather than flexibility. One of the aims of the project was to explore the potential for cheap multi-processor systems, constructed from commodity single board microprocessors interconnected via a standard backplane bus. Each microprocessor board contained local memory and an additional bus-connected memory-board providing shared global memory. Given the limited global resources, the designers focused on structuring the system more like independent local operating system instances. This would prove to be a precursor of later work like Hurricane, which attempted to apply distributed system principles to the problem of constructing a shared memory multi-processor operating system [134]. Although limited in nature, Tunis was one of the first operating systems to provide uniprocessor UNIX compatibility while employing a novel internal structure.

The early 1980's not only saw the emergence of tightly coupled shared memory multi-processor systems such as the CM* [100], but also loosely coupled distributed systems composed of commodity workstations interconnected via local area networking. Projects such as V [32] and Accent [107] attempted to provide a unified environment for constructing software and managing the resources of a loosely coupled distributed system. Unlike the operating systems for the emerging shared memory multi-processors, operating systems for distributed systems could not rely on hardware support for sharing. As such, they typically were constructed as a set of autonomous light-weight independent uniprocessor OS's which cooperated via network messages to provide a loosely coupled unified environment. Although dealing with very different performance tradeoffs, the distributed systems work influenced and intertwined with SMP operating systems research over the years. For example one of the key contributions of V [32] was micro-kernel support of light-weight user-level threads that were first-class and kernel visible.

In the mid 1980's, the Mach operating system was developed at Carnegie Mellon University based on the distributed systems Rig and Accent [16, 66, 107, 108, 145]. One of the key factors in Mach's success was the early commitment to UNIX compatibility while supporting user-level parallelism. In spirit, the basic structure of Rig, Accent and Mach is similar to Hydra and StarOS. All these systems are built around a fundamental IPC (Inter-Process Communication) model. For example, in the case of Mach, the basic IPC primitives are ports and messages. Processes provide services via ports to which messages are sent using capabilities and access rights. Mach advocates an object oriented-like model of services which are provided/located in servers. Rashid states

that Mach was, "designed to better accommodate the kind of general purpose shared-memory multi-processors which appear to be on their way to becoming the successors of traditional general purpose uniprocessor workstations and timesharing systems" [107]. Mach's main contribution with respect to multi-processor issues was its user-level model for fine-grain parallelism via threads and shared memory within the context of a UNIX process. This model became the standard model for user level parallelism in most UNIX systems. Otherwise, Mach takes the traditional approach of fine-grain locking of centralized data structures to improve concurrency on multi-processors[1]. The later Mach work does provide a good discussion of the difficulties associated with fine-grain locking, discussing issues of existence, mutual exclusion, lock hierarchies and locking protocols [16].

Another system developed in the 1980's, which was specifically designed for UNIX compatibility and multi-processor support, was the Topaz system at Digital Research [91, 92, 131]. Topaz was structured as a micro-kernel and a set of independent system servers. All communication was done with a Remote Procedure Call IPC provided by the micro-kernel. Topaz supported two types of user address spaces. One was a single threaded address space with an execution environment compatible with Ultrix, Digital's version of UNIX. The second was a Topaz custom address space, which additionally supported multiple threads of execution. A key research contribution of Topaz was to study the support required for multiple threads within a single UNIX process and its implications on the UNIX system interface. The Topaz system and Topaz-specific applications were developed in Modula2+ [111]. Modula2+ had explicit language support for concurrency, including: explicit thread support, explicit locking support, reference counted pointers, and a trace and sweep garbage collector. The Topaz authors report that the use of Modula2+'s support for concurrency and its object-oriented like encapsulation were particularly beneficial for systems construction. They state that on a 5-processor Topaz system, ideal scalability was rarely achieved. They note that for one file system copy application, a factor of 4.7 speedup was achieved, and that for parallel compilation a factor of 2 to 3 was achieved. No details are given in the literature pertaining to the scalability of Topaz. It is worth noting that Topaz provided UNIX compatibility while pursuing multi-processor support with an alternate system structure. Tornado and K42 have similar goals but with a greater focus on studying the structures required for scalability.

Gottlieb et al. [55] present operating system-specific synchronization techniques based on replace-add hardware support. The techniques attempt to increase concurrency by avoiding the use of

[1]Industrial systems such as Sequent's Dynix [13,52,65], one of Mach's contemporaries, employed fine-grain locking and exploring its challenges.

traditional lock or semaphore-based critical sections. The techniques they propose are generalized in later work by Herlihy [60,61]. Edler et al. [43] in the Symunix II system claim to have used the techniques of Gottlieb et al. as part of their design for supporting large-scale shared memory multiprocessors. Unfortunately, it is not clear to what extent the implementation of Symunix II was completed or to what extent the non-blocking techniques were applied. The main focus of Edler's work [43] was on Symunix II's support for parallelism in a UNIX framework, and specifically issues of parallel virtual memory management.

The later work of Massalin [86–88] explicitly studies the elimination of all locks in system software via the use of lock-free techniques. Massalin motivates the application of lock-free techniques for operating systems, pointing out some of the problems associated with locking:

Overheads: Spin locks waste cycles and blocking locks have costs associated with managing the queue of waiters.

Contention: Lock contention on global data structures can cause performance to deteriorate.

Deadlocks: Care must be taken to avoid deadlocks and this can add considerable complexity to the system.

Priority Inversion: Scheduling anomalies associated with locking, especially for real-time systems, introduce additional complexity.

One of the key methods used for applying lock-free techniques was the construction of system objects which encapsulated a standard data structure and associated synchronization implemented with lock free techniques. Massalin argues that such an encapsulation enables the construction of a system in which the benefits of lock free techniques can be leveraged while minimizing the impact of its complexity. Despite showing that the lock-free data structures have better raw uniprocessor performance, with respect to instructions and cycles, compared to versions implemented with locks, scalability is not established. Massalin's work was done in the context of the Synthesis operating system on a dual processor hardware platform, so the degree of parallelism studied was very limited. Furthermore, although the Synthesis work advocates reducing the need for synchronization, there is little guidance given or emphasis placed on this aspect[2]. The lock-free structures developed

[2]Primarily, two approaches are advocated; 1) Code Isolation and 2) Procedure Chaining. Massalin argues for the specialization of code paths which operate on independent data in a single threaded fashion thus avoiding the need for synchronization. This approach however, is explored in a limited fashion in Synthesis and relies on run-time code generation. Tornado's object-oriented structure, on which K42 is based, explores the parallel advantages of independent data in a more generalized and structured manner [49]. Procedure Chaining, the enqueueing of parallel work, does not improve concurrency or reduce sharing but simply enforces serialization via a scheduling discipline.

do not in themselves lead to less sharing or improved locality and hence, although having better performance than lock based versions, the large scale benefits are likely limited.

Although the scalability of lock-free techniques is not obvious, the work does add evidence for the feasibility of constructing an operating system around objects which encapsulate standard data structures along with synchronization semantics. Such an approach enables the separation of concerns with respect to concurrency, system structure and reuse of complex parallel optimizations in the Synthesis operating system.

The MOSIX researchers have similarly observed the need to limit and bound communication when tackling the problems of constructing a scalable distributed system [10, 11]. MOSIX focuses on the scalability issues associated with scaling a single UNIX system image to a large number of distributed nodes. The MOSIX work strives to ensure that the design of the internal management and control algorithms impose a fixed amount of overhead on each processor, regardless of the number of nodes in the system. Probabilistic algorithms are employed to ensure that all kernel interactions involve a limited number of processors and that the network activity is bounded at each node.

Unlike many of the other distributed systems, MOSIX is designed around a symmetric architecture where each node is capable of operating as an independent system, making its own control decisions independently. Randomized algorithms are used to disseminate information such as load without requiring global communication that would inhibit scaling. This allows each node to base its decisions on partial knowledge about the state of the other nodes without global consensus.

Although MOSIX is targeted at a distributed environment with limited sharing and coarse-grain resource management, its focus on limiting communication and use of partial information to ensure scalability is worth noting. Any system which is going to scale in the large must avoid algorithms that require global communications and leverage partial or approximate information where possible.

While studying the performance of Sprite, Ousterhout et al. independently observed difficulties associated with constructing a scalable operating system kernel [59]. Like Mach [16, 66, 107, 108, 145], the Sprite kernel, although designed for distributed systems, was designed to execute on shared memory multi-processor nodes of a network. It employed both coarse and fine-grain locking. The researchers at Berkeley conducted a number of macro and micro benchmarks to evaluate the scalability of Sprite on a five processor system. They made the following observations:

- The system was able to demonstrate acceptable scalability for the macro benchmarks up to

the five processors tested. Considerable contention was experienced in the micro benchmarks however, indicating that macro benchmark results will not extend past seven processors.

- Even a small number of coarse-grain locks can severely impact performance. A running Sprite kernel contains 500 to 1000 locks, but consistently two coarse-grain locks suffered the most contention and were primary limiting factors to scalability. At five processors, a coarse-grain lock was suffering 70% contention (i.e.,70% of attempts to acquire the lock failed).

- Coarse-grain locks are a natural result of incremental development and locking. Developers, when first implementing a service, will acquire the coarsest grain lock in order to avoid synchronization bugs and simplify debugging, and only subsequently do they split the locks to obtain better scalability.

- Lock performance is difficult to predict even for knowledgeable kernel developers who designed the system. The Sprite developers found that locks which they expected to be problematic were not and unexpected locks were. Further, performance was brittle with performance cliffs occurring at unpredictable thresholds; for example, good performance on five processors and poor performance on seven.

- Ousterhout et al. advocate better tools to help developers understand and modify locking structures.

In 1991 Cheriton et al. proposed an aggressive distributed shared memory parallel hardware architecture called Paradigm and also described OS support for it based on multiple co-operating instances of the V micro-kernel, a simple hand-tuned kernel designed for distributed systems construction, with the majority of OS function implemented as user-level system servers [31]. The primary approach to supporting sharing and application coordination on top of the multiple micro-kernel instances was through the use of a distributed file system. The authors state abstractly that kernel data structures, such as dispatch queues, are to be partitioned across the system to minimize interprocessor interference and to exploit efficient sharing, using the system's caches. Furthermore, they state that cache behavior is to be taken into account by using cache-friendly locking and data structures designed to minimize misses with cache alignment taken into consideration. Finally they also assert that the system will provide UNIX emulation with little performance overhead. It is unclear to what extent the system was completed.

In 1994, as part of the Paradigm project, an alternative OS structure dubbed the Cache Kernel was explored by Cheriton et al. [30]. At the heart of the Cache Kernel model was the desire to provide a finer grain layering of the system, where user-level application kernels are built on top of a thin cache kernel which only supports basic memory mapping and trap reflection facilities via an object model. From a multi-processor point of view, however, its architecture remained the same as the previous work, where a cluster of processors of the Paradigm system ran a separate instance of a Cache Kernel. The authors explicitly point out that such an architecture simplifies kernel design by limiting the degree of parallelism that the kernel needs to support. The approach results in reduced lock contention and eliminates the need for pursuing aggressive locking strategies. Cheriton et al. also allude to the fact that such an approach has the potential for improving robustness via limiting the impact of a fault to a single kernel instance and only the applications that depend on it. Although insightful, the Cache Kernel work did not explore or validate its claims. Hurricane [134] and Hive [27], contemporaries of the Cache Kernel did explore these issues in greater detail, adopting the high-level architecture proposed by the Paradigm group.

To summarize the experiences of the early multi-processor operating systems research, it is worthwhile reviewing the experiences of the RP3 researchers [18], who attempted to use the Mach micro kernel to enable multi-processor research. The RP3 authors state that a familiar programming environment was a factor in choosing Mach as it was BSD Unix compatible while it still promised flexibility and multi-processor support. The RP3 hardware was designed to minimize contention in the system by providing hardware support for distributing addresses of a page across physical memory modules, under control of the OS. There was no hardware cache coherence, but RP3 did have support for specifying uncached access on a page basis and user mode control of caches, including the ability to mark data for later software controlled eviction. The OS strategy was to start with the Mach micro-kernel, which supported a standard UNIX personality on top of it, and progressively extend it to provide gang scheduling, processor allocation facilities and the ability to exploit the machine-specific memory management features.

The RP3 researchers found that they needed to restructure the Mach micro kernel and UNIX support in order to utilize the hardware more efficiently and in order to improve performance, specifically needing to reduce memory contention. Some interesting points made include:

- Initially, throughput of page faults on independent pages degraded when more than three processors touched new distinct pages because of contention created by the spin lock algorithm used in the page-fault handling subsystem. Spin locks create contention in the memory

modules in which the lock is located and performance worsens with the number of spinners. "Essentially, lock contention results in memory contention that in turn exacerbates the lock contention." [25] By utilizing memory interleaving, it was possible to distribute data, in effect reducing the likelihood of co-locating lock and data and hence improving performance.

- Contention induced by slave processors spinning on a single shared word degraded boot performance to 2.5 hours. Eliminating the contention reduced boot time to 1/2 hour.
- The initial use of a global free-list did not scale, distributed per processor free-lists were introduced to yield efficient memory allocation performance.

The RP3 authors found bottlenecks in both the UNIX and Mach code with congestion in memory modules being the major source of slowdown. To reduce contention, the authors used hardware specific memory interleaving, low contention locks and localized free lists. We contend that the same benefits could have been achieved if locality had been explicitly exploited in the basic design.

As stated earlier, UNIX compatibility and performance were critical to the RP3 team. In the end, the flexibility provided by Mach did not seem to be salient to the RP3 researchers. Mach's internal traditional shared structure limited performance and its flexibility did not help to address these problems.

Based on the work covered in this section, we note that high performance multi-processor operating systems should:

1. enable high performance not only for large scale applications but also for standard UNIX workloads which can stress traditional implementations, and
2. avoid contention and promote locality to ensure scalability.

2.2 Distributed Data Structures and Adaptation

In this section we focus on the research related to the use of distributed data structures and associated work on adaptation.

2.2.1 Distributed Data Structures

A number of systems proposed the use of distributed data structures, albeit with various motivations. In this section we review some of the more prominent systems related work.

Distributed Systems: FOs and DSOs

Fragmented Objects(FOs) [17, 83, 120] and Distributed Shared Objects(DSOs) [8, 62, 136] both explore the use of a partitioned object model as a programming abstraction for coping with the latencies in a distributed network environment. Fragmented Objects represent an object as a set of fragments which exist in address spaces distributed across the machines of a local area network, while the object appears to the client as a single object. When a client invokes a method of an object, it does so by invoking a method of a fragment local to its address space. The fragments, transparently to the client, communicate amongst each other to ensure a consistent global view of the object. The Fragmented Objects work focuses on how to codify flexible consistency protocols within a general framework.

In the case of Distributed Shared Objects, distributed processes communicate by accessing a distributed shared object instance. Each instance has a unique id and one or more interfaces. In order to improve performance, an instance can be physically distributed with its state partitioned and/or replicated across multiple machines at the same time. All protocols for communication, replication, distribution and migration are internal to the object and hidden from clients. A coarse-grain model of communication is assumed, focusing on the nature of wide area network applications and protocols such as that of the World Wide Web. For example, global uniform naming and binding services have been addressed.

Clustered Objects are similar to FO's and DSO's in that they distribute/replicate state but hide this from clients by presenting the view of a single object. Clustered Object representatives correspond to FO's fragments.

Language Support: CAs and pSather

Chien et al. introduced Concurrent Aggregates(CAs) as a language abstraction for expressing parallel data structures in a modular fashion [33]. This work is concerned with the language issues of supporting a distributed parallel object model for efficient construction of parallel applications in a message passing environment. Similar to several concurrent object-oriented programming systems, an Actor model [1] is adopted. In this model, objects are self-contained, independent components of a computing system that communicate by asynchronous message passing. Such models typically impose a serialization of message processing by the use of message queues, thus simplifying the programmer's task by eliminating concurrency issues internal to an object. Chien et al. studied

a language extension called an *Aggregate* that permits object invocation to occur in parallel. An instance of an Aggregate has a single external name and interface, however each invocation is translated by a run-time environment to an invocation on an arbitrary representative of the Aggregate. The number of representatives for an Aggregate is declared by the programmer as a constant. Each representative contains local instances of the Aggregate fields. The language supports the ability for one representative of an Aggregate to name/locate and invoke methods of the other representatives in order to permit scatter and gather operations via function shipping and more complex cooperation.

pSather explores language and associated run-time extensions to support data distribution on NUMA multi-processors for Sather, an Eiffel-like research language [80]. Specifically, pSather adds threads, synchronization and data distribution to Sather. Unlike the previous work discussed, pSather advocates orthogonality between object orientation and parallelism; it introduces new language constructs independent of the object model for data distribution. Unlike Chien's Concurrent Aggregates, it does not impose a specific processing/synchronization model, nor does it assume the use of system-provided consistency models/protocols like Distributed Shared Objects or Fragmented Objects. In his thesis, Chu-Cheow proposes two primitives for replicating reference variables and data structures such that a replica is located in each cluster of a NUMA multi-processor [80]. Since the data distribution primitives are not integrated into the object model, there is no native support for hiding the distribution behind an interface. There is also no support for dynamic instantiation or initialization of replicas, nor facilities for distributed reclamation. Note that this work assumes that every replicated data element will have a fixed mapping of one local representative for every cluster with respect to the hardware organization and that initialization will be done for all replicas once prior to the data element's use.

The pSather work explores the advantages of several distributed data structures built on top of the pSather primitives. This was done in the context of a set of data parallel applications on a NUMA multi-processor. Given the regular and static parallelism of data parallel applications, the semantics of the data structures explored are limited and restricted. The data structures considered include: a workbag (a distributed work queue), a distributed hash table, a distributed matrix and a distributed quad tree. The primary focus is to validate pSather implementations of various applications; this includes the construction of the distributed data structures and the applications themselves. Although the primary focus of the pSather work is on evaluating the overall programmer experience, using the parallel primitives and the library of data structures

developed, the performance of the applications is also evaluated with respect to scale. The author highlights some of the tradeoffs in performance with respect to remote accesses given variations in the implementation of the data structures and algorithms of the applications. He points out that minimizing remote accesses, thus enhancing locality, is key to good performance for the applications studied. Unfortunately, it appears that the author does not compare the performance of the distributed data structures to centralized implementations, so it is difficult to get a feeling for the exact benefits of distribution.

Finally, it is not clear that the distributed data structures that are explored in the pSather work are appropriate for systems software, which is dynamic and event-driven in nature, as opposed to the regular and static parallelism of scientific applications. For example, consider the semantics of the distributed hash table:

- The state stored in the hash table is monotonically increasing; once inserted, an item will never be removed.
- Each local hash table employs coarse-grain locking.
- There are no facilities for modifying stored state.
- There are no facilities for hashing distributed state.

The restrictions are not necessarily problematic for data parallel applications which would utilize such a hash table to publish progressive results of a large distributed calculation. However, it would be difficult to use such a hash table as a cache of page descriptors for the resident memory pages when implementing an operating system. Systems code, in general, cannot predict the parallel demand on any given data structure instance, and hence the data structure creation, initialization, sizing and concurrency must be dynamic in nature. Additionally, systems software must be free to exploit aggressive optimizations which simple semantics may not permit. For example, once it is determined that a distributed hash table is going to be used, there may be other optimizations that present themselves by distributing the data fields of the elements that are being stored. In the case of page descriptors, rather than just storing a reference to a page descriptor, in the local replicas of the hash table, one might want to allow each replica to store local versions of the access bits of the page descriptor in order to avoid global memory access and synchronization on the performance critical resident page fault path.

Topologies and DSAs

In the early 1990's there was considerable interest in message passing architectures, which typically leveraged a point to point interconnection network and the promise of unlimited scalability. Such machines, however, did not provide a shared memory abstraction. They were typically used for custom applications, organized as a collection of threads which communicate via messages, aware of and tuned for the underlying interconnection geometry of the hardware platform.

In an attempt to ease the burden and generalize the use of such machines, Bo et al. proposed OS support for a distributed primitive called a Topology [114]. A Topology attempts to isolate and encapsulate the communication protocol and structure among a number of identified communicating processes via a shared object-oriented abstraction.A Topology's structure is described as a set of vertices and edges where the vertices are mapped to physical nodes of the hardware and edges capture the communication structure between pairs of nodes corresponding to vertices. An example would be an inverse broadcast which encapsulates the necessary communication protocol to enable the aggregation of data from a set of distributed processes. The Topology is implemented to minimize the number of nonlocal communications for the given architecture being used. They are presented as heavy-weight OS abstractions requiring considerable OS support for their management and scheduling. Applications make requests to the operating system to instantiate, configure and use a Topology. Each type encapsulates a fixed communication protocol predefined by the OS implementors. Applications utilize a Topology by creating, customizing and invoking an instance, binding the application processes to the vertices, specifying application functions for message processing, specifying the state associated with each vertex and invoking the specified functions by sending messages to the instance.

In subsequent work, motivated by significant performance improvements obtained with distributed data structures and algorithms on a NUMA multi-processor for Traveling Sales Person (TSP) programs [99], Clémençon et al. proposed a distributed object model called Distributed Shared Abstractions (DSAs) [34,35]. This work is targeted at increasing the scalability and portability of parallel programs via a reusable user level library which supports the construction of objects that encapsulate a DSA. Each object is composed of a set of distributed fragments similar to the Fragmented Objects and Distributed Shared Objects discussed earlier. The run-time implementation and model are, however, built on the previous work on Topologies.

Akin to the work in pSather, Clémençon et al. specifically cited distribution as a means for

improving performance by improving locality and reducing remote accesses on a multi-processor. The authors asserted the following benefits:

- potential reduction in contention when accessing an object in parallel, since many operations on the object will access only locally stored copies of its distributed state,
- decreases in invocation latencies, since local accesses are faster than remote accesses, and
- the ability to implement objects so that they may be used on both distributed and shared memory platforms, therefore increasing the portability of applications using them.

As implied by the last two points, and influenced by their earlier work, Clémençon et al. assumed a message passing communication model between fragments. Despite the message passing focus, the performance results and analysis presented are relevant to the work done by our research group at the University of Toronto [3–5, 48, 50].

Clémençon et al. observed two factors which affect performance of shared data structures on a NUMA multi-processor:

1. contention due to concurrent access (synchronization overhead), and
2. remote memory access costs (communication overhead).

They observed that distribution of state is key to reducing contention and improving locality. When comparing multiple parallel versions of the TSP program, they found that using a centralized work queue protected by a single spin lock limited speedup to 4 times whereas a 10 times speed up was possible with a distributed work queue on a system with 25 processors. Further, they found that, by leveraging application specific knowledge, they were able to specialize the distributed data structure implementation to further improve performance. They demonstrated that despite additional complexities, distributed implementations with customized semantics can significantly improve application performance. Note that, in the results presented, Clémençon et al. do not isolate the influence of synchronization overhead versus remote access in their results[3].

Based on performance studies of TSP on two different NUMA systems, Clémençon et al. state that, "Any large scale parallel machine exhibiting NUMA memory properties must be used in a fashion similar to distributed memory machines, including the explicit distribution of the state and functionality of programs' shared abstractions." [35] They further note that machines with

[3]Concurrent centralized implementations which employ fine-grain locking or lock free techniques were not considered.

hardware cache coherence, such as the KSR [46], do not alleviate the need for distribution. Cache coherence overheads further enforce the need for explicit distribution of state and its application-specific management, in order to achieve high performance. Based on initial measurements done on small scale SGI multi-processors, Clémençon et al. predicted that, given the trends in the disparity between processor speeds and memory access times, distributed data structures will be necessary even on small scale multi-processors. They point out an often over-looked aspect of the use of shared memory multi-processors:

> Multi-Processors are adopted for high performance and shared memory multi-processors are claimed as superior to message passing systems as they offer a simple convenient programming model. However, to achieve high performance on shared memory multi-processors, they require the use of complex distributed implementations akin to those used on message passing systems.

This mirrors our own experience in using shared-memory multi-processors and motivates our Clustered Object work, which attempts to limit the impact of the added complexity through encapsulation and reuse.

The actual detailed model and implementation of the DSA run-time appears to have been strongly influenced by: 1) the previous Topology work, 2) assumptions about application use, and 3) the desire to be portable to distributed systems. This resulted in a heavy-weight facility which is only appropriate for coarse-grain, long-lived, fully distributed application objects whose access patterns are well known. Limiting characteristics include:

- expensive message based interface to objects where object access is 6 times more expensive than procedure invocation,
- a restricted scheduling and thread model,
- support only for inter-fragment communication via remote procedure calls and no support for direct shared memory inter-fragment access,
- expensive binding operations that preclude short lived threads,
- no support for efficient allocation or deallocation of objects, and
- manual initialization that is serial in nature.

The use of the DSA library seems to have been very limited with only the distributed work queue explored in the context of the TSP application. Given limited use and lack of experience in a distributed systems environment, it is unclear if the restrictive, fully distributed message passing model is justified.

In summary there are three key results from the DSA work:

1. A shared memory multi-processor parallel application's performance can greatly benefit from the use of distributed data structures.

2. Better performance can be achieved by exploiting application-specific knowledge to tailor distributed implementations, as opposed to generic object implementations or general distributed methodologies which impose fixed models for the sake of generality.

3. Locality optimization can be effectively employed at the level of abstract data types.

Clustered Objects

In 1995, our research group at the University of Toronto proposed the use of Clustered Objects [101] to encapsulate distributed data structures for the construction of NUMA multi-processor systems software. Like the Concurrent Aggregates, Fragmented Objects, Distributed Shared Objects and Distributed Shared Abstractions, Clustered Objects enable distribution within an object-oriented model with the motivation to enable NUMA locality optimizations, like those advocated by pSather and DSAs.[4] Unlike the previous approaches, Clustered Objects were designed for systems level software and uniquely targeted at ubiquitous use, where every object in the system is a Clustered Object with its own potentially unique structure. They are designed to support both centralized and distributed implementations, fully utilizing the hardware support for shared memory where appropriate. They do not impose any constraints on the access model or computational model either externally or internally to an object. The supporting mechanisms are light-weight and employ lazy semantics to ensure high performance for short-lived system threads. Clustered Objects are a key focus of this dissertation and are discussed in more detail throughout this document.

[4] Based on the experience of developing two multi-processor systems and associated systems software, the University of Toronto group independently came to similar observations and conclusions to that of the DSA authors [34, 35, 99], identifying the importance of locality to shared memory multi-processor performance and the possibility of utilizing distribution to improve locality.

Distributed Hash Tables

Finally, there has been a body of work which has looked at the use of distributed hash tables in the context of specific distributed applications, including distributed databases [44], cluster based internet services [57], peer-to-peer systems [38] and general distributed data storage and look up services [24]. In general, the work done on distributed data structures for distributed systems is primarily concerned with the exploration of the distributed data structure as a convenient abstraction for constructing network based applications, increasing robustness via replication. Like the Fragmented Object and Distributed Shared Object work, the use of distributed hash tables seeks a systematic way of reducing and hiding network latencies. The coarse-grain nature and network focus of this type of work results in few insights for the construction of performance critical and latency sensitive shared memory multi-processor systems software.

2.2.2 Adaptation for Parallelism

Motivated by the sensitive and dynamic demands of parallel software, a number of researchers have proposed leveraging the strict modularity imposed by object orientation in order to facilitate adaptability and autonomic systems/computing. Unlike the previous work discussed, which focused on object orientation for the sake of flexibility in system composition and configuration, the work on adaptation focuses on using the encapsulation enforced by object boundaries to isolate the computations which can have sensitive parallel performance profiles. Adaptation is enabled by introducing mechanisms which allow reconfiguration of the isolated components, either by adjusting parameters of the component or complete replacement. The most appropriate configuration can then be chosen in order to maximize the performance based on the current demands. There has been considerable work in the area of adaptable or autonomic computing. We restrict our discussion to the work on adaptation for the sake of parallel performance.

In CHAOSarc [54], the authors explore a parallel real-time system for robot control using an object-oriented decomposition. The work studies predictability within the context of a highly configurable parallel programming environment. The focus on predictability led to a fine-grain classification of computation and locking semantics in order to be able to match application demand. This resulted in a large configuration space in which the components used to implement a computation must match the run-time constraints and aspects of the computation. This lead Mukherjee et al. to consider reconfiguration of objects and the use of adaptive objects [99].

Mukherjee et al. [99] explore the costs, factors and tradeoffs with building adaptive systems in the context of implementing parallel solutions to the Traveling Sales Person problem. They attempt to construct and formalize a theory of adaptation, proposing a categorization of objects as non-configurable, reconfigurable, and adaptable. Reconfigurable objects permit external changes to mutable properties which affect their operation. Adaptable objects encapsulate a reconfigurable object, monitoring facility, adaptation policy and reconfiguration mechanism. The utility of the formalization proposed is unclear and no concrete implementation or mechanisms for supporting it is given. The Mukerjee et al. do provide strong motivation with respect to parallel performance of the TSP applications. They illustrate the benefits of an adaptive lock object which modifies its spin and blocking behavior based on the demands it experiences versus using a static lock object. A 17% improvement in application execution time was observed when studying a centralized algorithm of TSP which uses shared data structures and a 6.5% improvement when considering a distributed implementation of TSP.

Motivated by the results above, the Gheith et al. attempted to extend the single application benefits observed to the entire system by enabling adaptation in the system layers. They proposed an architecture for a reconfigurable parallel micro-kernel called KTK [53]. They assert that:

- Run-time behavior differs across multiple applications and across multiple phases of a single application.
- Operating system kernel configurations can provide high-performance applications with the policies and mechanisms best suited to their characteristics and to their target hardware. Gheith et al. appeal to the standard flexibility arguments of previous object-oriented systems.
- Dynamic kernel reconfiguration can improve performance by satisfying each application's needs.
- Efficient application state monitoring can detect changes in application requirements which are not known prior to program execution time.

The KTK architecture proposes a number of core kernel components which support configuration, reconfiguration and adaptation. Each object identifies a set of attributes which are mutable and support some form of arbitration with respect to the attribute change. The micro-kernel should provide monitoring of kernel components in order to facilitate adaptation policies and also provide support for application-defined adaptation policies. Although the case for system reconfiguration

is compelling, it is unclear to what extent the KTK prototype achieved the goals set out or to what extent application performance was improved or facilitated. Motivated by KTK, Silva et al. attempt to factor the support for reconfiguration of a parallel object-based software into a library called CTK [122] in order to facilitate a programming model that enables the expression and implementation of program configuration and the run-time support for performance improvements by changing configuration. Building on the KTK model, CTK adopts a model which incorporates specification of mutable attributes and policies for attribute change. It also supports efficient online capture of performance data in order to enable dynamic configuration. Utilizing the group's previous work, CTK explores a distributed work queue DSA (see 2.2.1) with respect to reconfiguration in Traveling Sales Person solutions. The work proposes the use of a custom language that incorporates the expression of object attributes and other facilities. CTK focuses on expressive and general support for the specification of dynamic policies and attribute change. Evaluation was limited to a single user-level application and the suitability of CTK in an operating system is unclear.

Based on similar motivations, the K42 group has explored the integration of mechanisms for enabling run-time adaptation of Clustered Objects [4, 5, 63, 64, 124]. Unlike the KTK and CTK work, the K42 work focuses on the mechanism for enabling efficient replacement of a distributed object rather than general issues of adaptation. The K42 work is uniquely focused on ensuring low overheads so that object replacement can be used pervasively in the systems software. Furthermore, the K42 work explores the use of adaptation with respect to system objects which are in critical operating system paths and are subject to unpredictable dynamic concurrent access.

2.2.3 Summary

Previous work on distributed data structures and adaptation suggests that a high performance operating system should:

- Enable data distribution, as highly concurrent shared memory multi-processor software requires distributed data structures to ensure high concurrency and low latency. Since operating systems must reflect the concurrency of the workload, they, by definition, need to enable the highest possible degree of concurrency in order not to limit application performance.

- Utilize object orientation to help cope with the complexity introduced by data distribution, but ensure low overhead.

- Support adaptation in order to support variability in parallel workloads and OS demands.

2.3 Modern Multi-Processor Operating Systems Research

There have been a number of papers published on performance issues in shared-memory multi-processor operating systems, but mostly in the context of resolving specific problems in a specific system [21, 26, 30, 90, 106, 129]. These operating systems were mostly for uniprocessor or small-scale multi-processor systems, trying to scale up to larger systems. Other work on locality issues in operating system structure was mostly either done in the context of earlier non-cache-coherent NUMA systems [28], or, as in the case of Plan 9, was not published [105]. Two projects that were aimed explicitly at large-scale multi-processors were Hive [27], and Hurricane [134]. Both independently chose a clustered approach by connecting multiple small-scale systems to form either, in the case of Hive, a more fault tolerant system, or, in the case of Hurricane, a more scalable system. However, both groups ran into complexity problems with this approach and both have moved on to other approaches; namely Disco [20] and Tornado [48], respectively.

2.3.1 Characteristics of Scalable Machines

SMP architectures present the programmer with the familiar notion of a single address space within which multiple processes exist, possibly running on different processors. Unlike a message-passing architecture, an SMP does not require the programmer to use explicit primitives for the sharing of data. Hardware-supported shared memory is used to share data between processes, even if running on different processors. Many modern SMP systems provide hardware cache coherence to ensure that the multiple copies of data in the caches of different processors (which arise from sharing) are kept consistent.

Physical limits, cost efficiency and desire for scalability have led to SMP architectures that are formed by inter-connecting clusters of processors. Each cluster typically contains a set of processors and one or more memory modules. The total physical memory of the system is distributed as individual modules across the clusters, but each processor in the system is capable of accessing any of these memory modules in a transparent way, although it may suffer increased latencies when accessing memory located on remote clusters. SMPs with this type of physical memory organization are called Non-Uniform Memory Access (NUMA) SMPs. Examples of such NUMA SMP architectures include Stanford's Dash [78] and Flash [74] architectures, University of Toronto's

Hector [140] and NUMAchine [139] architectures, Sequent's NUMA-Q [119] architecture and SGI's Cray Origin2000 [76]. NUMA SMPs that implement cache coherence in hardware are called CC-NUMA SMPs. In contrast, multi-processors based on a single bus have Uniform Memory Access times and are called UMA SMPs, but are limited in scale.

When discussing data access on a NUMA multi-processor it is convenient to appeal to an abstract notion of distance. An access to a data item is considered "near or close" if the data item is stored in a memory module which has lower latencies for a specific processor, given a NUMA architecture, and conversely the data item is "far or remote" if it is located in a module for which the latencies are higher for the accessing processor.

It can be difficult to realize the performance potential of a CC-NUMA SMP. The programmer must not only develop algorithms that are parallel in nature, but must also be aware of the subtle effects of sharing both in terms of correctness and in terms of performance. These effects include:

- Access to shared data can suffer increased communication latencies due to the coherence protocols and distribution of physical memory.

- The use of explicit synchronization is needed to ensure correctness of shared data, which can also induce additional computation and communication overheads.

- False sharing reduces the effectiveness of the hardware caches and results in the same high cache coherence overhead as true sharing. (False sharing occurs when independently accessed data is co-located in the same cache line and requires careful data layout in memory to avoid.)

Memory latencies and cache consistency overheads can often be reduced substantially by designing software that maximizes the locality of data accesses. Replication and partitioning of data are primary techniques used to improve locality. Both techniques allow processes to access localized instances of data in the common case. They decrease the need for remote memory accesses and lead to local synchronization points that suffer less contention.

Other more coarse-grain approaches for improving locality in general SMP software include automated support for memory page placement, replication and migration [75, 84, 138] and cache affinity aware process scheduling [42, 58, 85, 126, 137].

The two key factors affecting multi-processor software, and in particular OS performance, besides the policies and algorithms employed by the software, are memory system and locking behaviors. The key to maximizing memory system performance on a multi-processor is to minimize the

amount of (true and false) sharing, particularly for read-write data structures. Not paying careful attention to sharing patterns can cause excessive cache coherence traffic, resulting in potentially terrible performance due to the direct effect of the extra cache misses and to the secondary effect of contention in the processor-memory interconnection network and at the memory itself. 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 [132]. Similarly, Rosenblum noted in a study of an eight 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 in the IRIX operating system [110].

In larger systems, the secondary effects become more significant. Moreover, in large NUMA systems, it is also necessary to take memory access latencies into account, considering that accesses to remote memory modules can cost several times as much as accesses to local memory modules. The significance of this was observed by 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 [134].

The sharing of cache lines can often be reduced by applying various replication and partitioning strategies, whereby each processor (or set of processors) is given a private copy or portion of the data structure. The same strategy also helps increase locality, aiding larger NUMA systems. However, replication and partitioning requires more work in managing and coordinating the multiple data structures.

Despite disputes about the details, it is widely accepted that scalable large multi-processor hardware is realizable, given a hardware-supported distributed shared memory architecture. But such hardware will have properties that require special attention on the part of systems software if general purpose workloads are to be supported.

2.3.2 Operating Systems Performance

Poor performance of the operating system can have considerable impact on application performance. For example, for parallel workloads studied by Torrellas et al., the operating system accounted for as much as 32% to 47% of the non-idle execution time [132]. Similarly Xia and Torrellas showed that for a different set of workloads, 42% to 54% of the time was spent in the operating system [143], while Chapin et al. found that 24% of total execution time was spent in the operating system [26] for their workload.

To keep the operating system from limiting application performance, it must be highly concurrent. The traditional approach to developing SMP operating systems has been to start with a uniprocessor operating system and to then successively tune it for concurrency. This is achieved by adding locks to protect critical resources. Performance measurements are then used to identify points of contention. As bottlenecks are identified, locks are split into multiple locks to increase concurrency, leading to finer-grained locking. Several commercial SMP operating systems have been developed as successive refinements of a uniprocessor code base. Denham et al. provides an excellent account of one such development effort [41]. This approach is ad hoc in nature, however, and leads to complex systems, while providing little flexibility. Adding more processors to the system, or changing access patterns, may require significant re-tuning.

The continual splitting of locks can also lead to excessive locking overheads. In such cases, it is often necessary to design new algorithms and data structures that do not depend so heavily on synchronization. Examples include the: software set associative cache architecture developed by Peacock et al. [102, 103], kernel memory allocation facilities developed by McKenny et al. [96], fair fast scalable reader-writer locks developed by Krieger et al. [73], performance measurement kernel device driver developed by Anderson et al. [2], and the intra-node data structures used by Stets et al. [128].

The traditional approach of splitting locks and selectively redesigning also does not explicitly lead to increased locality. Chapin et al. studied the memory system performance of a commercial Unix system, parallelized to run efficiently on the 64-processor Stanford DASH multi-processor [26]. They found that the time spent servicing operating system data misses was three times more than time spent executing operating system code. Of the time spent servicing operating system data misses, 92% was due to remote misses. Kaeli et al. showed that careful tuning of their operating system to improve locality allowed them to obtain linear speedups on their prototype CC-NUMA system, running OLTP benchmarks [67].

In the early to mid-1990's, researchers identified memory performance as critical to system performance [26, 29, 89, 110, 132]. They noted that cache performance and coherence are critical aspects of SMP hardware which must be taken into account by software, and that focusing on concurrency and synchronization is not enough.

Rosenblum et al. explicitly advocated that operating systems must be optimized to meet the demands of users for high performance [110]. However, they point out that operating systems are large and complex and the optimization task is difficult and, without care, tuning can result in in-

creased complexity with little impact on the end-user performance. The key is to focus optimization by identifying performance problems. They studied three important workloads:

1. Program development workload,
2. Database workload, and
3. Large simulations which stress the memory subsystem.

They predicted that, even for small scale SMP's, coherence overheads induced by communication and synchronization overheads would result in MP OS services consuming 30% to 70% more resources than uniprocessor counterparts. They also observed that larger caches do not help alleviate coherence overhead, so the performance gap between MP OSs and UP OSs will grow unless there is focus on kernel restructuring to reduce unnecessary communication. They pointed out that, as the relative cost of coherence misses goes up, programmers must focus on data layout to avoid false sharing, and that preserving locality in scheduling is critical to ensuring effectiveness of caches. Rescheduling processes on different processors can result in coherence traffic on kernel data structures.

The research at the University of Toronto has been addressing these same issues. Unlike many of the previously discussed MP OS research efforts, the University of Toronto chose to first focus on multi-processor performance, thereby uniquely motivating, justifying and evaluating the operating system design and implementation based on the structure and properties of scalable multi-processor hardware. Motivated by the Hector multi-processor [140], representative of the architectures for large scale multi-processors of the time [12, 46, 78, 104], the group chose a simple structuring for the operating system which directly mirrored the architecture of the hardware, hoping to leverage the strengths of the hardware structure while minimizing its weaknesses.

By focusing on performance rather than flexibility, the Hurricane group was motivated to acknowledge, analyze and identify the unique operating system requirements with respect to scalable performance. Particularly, based on previous literature and queuing theory analysis, the following guidelines were identified [134]:

Preserving parallelism: The operating system must preserve the parallelism afforded by the applications. If several threads of an executing application (or of independent applications running at the same time) request independent operating system services in parallel, then they must be serviced in parallel; otherwise the operating system becomes a bottleneck, limiting

scalability and application speedup. Critically, it was observed that an operating system is demand driven and its services do not utilize parallelism, thus parallelism can only come from application demand. Therefore, the number of operating system service points must increase with the size of the system and the concurrency available in accessing the data structures must grow with the size of the system to make it possible for the overall throughput to increase proportionally.

Bounded overhead: The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors. If the overhead of each service call increases with the number of processors, the system will ultimately saturate, so the demand on any single resource cannot increase with the number of processors. For this reason, system wide ordered queues cannot be used, and objects cannot be located by linear searches if the queue lengths or search lengths increase with the size of the system. Broadcasts cannot be used for the same reason.

Preserve locality: The operating system must preserve the locality of the applications. It is important to consider the memory access locality in large-scale systems because, for example, many large-scale shared memory multi-processors have non-uniform access (NUMA) times, where the costs of accessing memory is a function of the distance between the accessing processor and the target memory, and because cache consistency incurs more overhead in a large system. Specifically it was noted that locality can be increased a) by properly choosing and placing data structures within the operating system, b) by directing requests from the application to nearby service points, and c) by enacting policies that increase locality in the applications' memory accesses. For example, policies should attempt to run the processes of a single application on processors close to each other, place memory pages in proximity to the processes accessing them, and direct file I/O to devices close by. Within the operating system, descriptors of processes that interact frequently should lie close together, and memory mapping information should lie close to the processors which must access it to handle page faults.

Although some of these guidelines have been identified by other researchers [9,123], we are not aware of other general purpose shared memory multi-processor operating systems which pervasively utilize them in their design. Over the years, these guidelines have been refined but have remained a central focus of the body of research work done at the University of Toronto.

Hurricane, in particular, employed a coarse-grain approach to scalability, where a single large scale SMP was partitioned into clusters of a fixed number of processors. Each cluster ran a separate instance of a small scale SMP operating system, cooperatively providing a single system image. Hurricane attempted to directly reflect the hardware structure, utilizing a collection of separate instances of a small-scale SMP operating system, one per hardware cluster. Implicit use of shared memory is only allowed within a cluster, we refer to this approach as fixed clustering.. Any co-ordination/sharing between clusters occurs using a more expensive explicit facility. It was hoped that any given request by an application could in the common case be serviced on the cluster on which the request was made with little or no interaction with other clusters. The fixed clustering approach limits the number of concurrent processes that can contend on any given lock to the number of processors in a cluster. Similarly, it limits the number of per-processor caches that need to be kept coherent. The clustered approach also ensures that each data structure is replicated into the local memory of each cluster.

Despite many of the positive benefits of clustering, it was found that: (*i*) the traditional within-cluster structures exhibit poor locality, which severely impacts performance on modern multi-processors; (*ii*) the rigid clustering results in increased complexity as well as high overhead or poor scalability for some applications; (*iii*) the traditional structures as well as the clustering strategy make it difficult to support the specialized policy requirements of parallel applications [48].

Related work at Stanford on the Hive operating system [27] also focused on clustering, firstly as a means of providing fault containment and secondly as a means for improving scalability. Having experienced similar complexity and performance problems with the use of fixed clustering, the Stanford research group began a new project in the late 1990's called Disco [20, 56, 109]. The Disco project pursued strict partitioning as a means for leveraging the resources of a multi-processor. Rather than trying to construct a kernel which can efficiently support a single system image, they pursue the construction of a kernel which can support the execution of multiple virtual machines (VMs). By doing so, the software within the virtual machines is responsible for extracting the degree of parallelism it requires from the resources allocated to the VM on which it is executing. Rather than wasting the resources of a large scale machine on a single OS instance that is incapable of efficiently utilizing all the resources, the resources are partitioned across multiple OS instances. There are three key advantages to this approach:

1. The underlying systems software which enables the partitioning does not itself require high concurrency.

2. Standard workloads can be run by leveraging the Virtual Machine approach to run standard OS instances.

3. Resources of a large scale machine can be efficiently utilized with standard software, albeit without native support for large scale applications and limited sharing between partitions.

To some extent this approach can be viewed as a tradeoff which permits large scale machines to be leveraged using standard systems software.

Two fundamental points are raised by the Disco research:

1. Standard operating systems do not effectively support large scale multi-processors.

2. Despite point 1, the standard environment offered by commodity systems is compelling enough to justify partitioning of the hardware.

This implies that a new scalable system must support the standard operating environment of a commodity system if it is to be effective.

Like the Stanford group, the Toronto researchers also pursued a new project based on their experiences with fixed clustering. In contrast, however, the Toronto group chose to pursue an operating systems structure which relaxed the boundaries imposed by clustering when constructing its new operating system which called Tornado. The fundamental approach was to explore the structuring of an operating system kernel so that a single system image could be efficiently scaled without having to appeal to the fixed boundaries of clustering.

Like other systems, Tornado had an object-oriented design, but not primarily for the software engineering benefits or for flexibility, but rather for multi-processor performance benefits. More specifically, the design of Tornado was based on the observations that: (*i*) operating systems are driven by the request of applications for virtual resources; (*ii*) to achieve good performance on multi-processors, requests 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 on which they are issued. This is achieved in Tornado by adopting an object-oriented approach where each virtual and physical resource in the system is represented by an independent object, so that accesses on different processors to different objects do not interfere with each other. Details of the Tornado operating system are given elsewhere [48, 49].

The contributions of the Tornado work include:

1. an appropriate object decomposition for a multi-processor operating system,

2. the development of scalable and efficient support in the object run-time that would enable declustered (distributed) implementations of any object (Objects in Tornado were thus dubbed Clustered Objects),

3. the development of a semi-automatic garbage collection scheme incorporated in the object run-time system that facilitates localizing lock accesses and greatly simplifies locking protocols,[5] and

4. the development of a core set of low-level operating system facilities which are tuned for multi-processor performance showing high degrees of concurrency and locality.

The primary core set of low-level OS facilities focused on in Tornado included:

- Scalable efficient multi-processor memory allocation.

- Light-weight protection domain crossing which is focused on preserving locality, utilizing only local processor resources in the common case.

In Tornado, the majority of the system's objects did not utilize distribution. Gamsa's work on Clustered Objects in Tornado focused on developing the underlying infrastructure and basic mechanisms [48]. My work, developing the Clustered Object model and studying the development and use of distributed object implementations, began in Tornado [3] utilizing the supporting mechanisms developed by Gamsa.

In the late 1990's, IBM began an effort to develop K42, a research operating system to explore scalability and a novel user-level structure, while providing compatibility with a standard OS. In an attempt to account for scalability within the basic design and structure, IBM chose to base K42 on Tornado. The University of Toronto licensed Tornado to IBM. Our group at the University of Toronto has closely collaborated with IBM to design and implement K42, with our research group focusing on K42's scalability.

In this dissertation, I explore the use of distributed data structures in K42, continuing my initial work on Clustered Objects from Tornado. This includes the standardization of the use of the Clustered Object mechanisms proposed in Tornado via a set of protocols which provide

[5] With the garbage collection scheme, no additional (existence) locks are needed to protect the locks internal to the objects. As a result, Tornado's *locking strategy* results in much lower locking overhead, simpler locking protocols, and can often eliminate the need to worry about lock hierarchies. As part of Tornado's garbage collection scheme, the Toronto research group independently developed a lock free discipline similar to that of Read-Copy-Update [97].

a distributed object-oriented model, permitting incremental development while fully leveraging hardware-supported shared memory. Doing so, this work builds upon the lessons of previous research:

- focus on performance over flexibility,
- maximize concurrency, focusing on structures and algorithms rather than improved synchronization as done in earlier systems,
- maximize hardware locality,
- enable adaptation in order to cope with variability in parallel demands.

In addition to the development of the Clustered Object model and infrastructure, this dissertation illustrates the viability of using distributed data structures in the construction and optimization of key OS services, demonstrating considerable improvements in the scalability of standard OS workload benchmarks. Two key features of K42 that facilitate our work are:

1. an object-oriented decomposition, as advocated by the Tornado work, which reflects workload independence and enables per-resource instance optimization, and
2. a Linux compatible infrastructure that provides an accepted software environment, allowing the execution of standard workloads.[6]

In the next chapter we will look more carefully at the motivation for introducing the complexity of distribution into an operating system kernel.

[6]Previous work has shown that using a standard software environment is critical to ensuring relavancy of OS research performance results.

Chapter 3

Motivation and Clustered Object Overview

In this chapter, we motivate the use of distribution in the construction of systems software for shared memory multi-processor operating systems. We then provide an overview of Clustered Objects, a software construction for the systematic application of distribution in an object-oriented system. The aim of this chapter is to provide the necessary background for the following chapter which describes examples of distributed structures that have been implemented in K42 as a part of this work.

3.1 Motivation: Performance

To fully utilize multi-processor architectures, three issues require special attention:

Concurrency: Software must exploit concurrency to fully utilize the processing resources of an SMP. Concurrent processes use shared memory to cooperate, but concurrent updates to shared data must be serialized to ensure consistency. The addition of synchronization in the form of locks and other atomic primitives can be used to control concurrency. Deciding where to add synchronization and what type of synchronization to use can be non-trivial. A strategy that is too coarse can lead to highly contended locks and limited concurrency. On the other hand, a strategy that is too fine can lead to excessive overheads due to having to acquire and release many locks. Often, a complete redesign of an algorithm and its data structures can significantly reduce the amount of shared data and hence the need for synchronization.

Cache Misses: Efficient use of caches is critical to good performance for two reasons. Firstly, a low cache miss rate ensures that processors do not spend large amounts of time stalling on memory accesses. Secondly, it reduces the traffic on shared system busses. With per-processor caches, processors accessing data on the same cache line, either because the data is being shared directly or because it is being shared falsely, causes the line to be replicated into multiple caches. Sharing of cache lines by two or more processors causes an increase in consistency overhead and an increase in cache misses[1]. Avoiding shared data and carefully laying out data in memory to avoid false sharing can substantially reduce cache line sharing, and the associated increase in overheads.

Remote Memory Access: To achieve good performance on Non-Uniform Memory Access (NUMA) systems, the extra costs of remote memory accesses must be avoided. Caches can help to reduce the cost of remote accesses, but do not eliminate the costs completely. The first access to a remote data element has a higher cost. Additionally, true and false read-write sharing can force invalidation of locally cached copies of remote data. Eliminating shared data and carefully placing data in the memory modules closest to the processors that access the data can reduce the number of remote memory accesses.

We define the term *locality management* to refer to the combination of increasing concurrency, reducing cache misses and reducing remote memory accesses. Gamsa et al. have outlined a set of design principles for developing software that manages locality [50], the main points of which are:

- Concurrency
  - Replicate read locks and implement write locks as a union of the read locks. This increases concurrency by making the locks finer grained.
- Cache Misses
  - Segregate read-mostly data from frequently modified data to reduce misses due to false sharing.
  - Segregate independently accessed data to eliminate false sharing.
  - Replicate write-mostly data to reduce sharing.

[1] To be more precise, invalidation-based cache coherence protocols require that a processor writing to a line obtain ownership of the line if it does not already own it (upgrade miss). This results in the invalidation of all copies of the line in other processors' caches. Thus all other processors will suffer a miss (sharing miss) on a subsequent access to the line.

  - Use per-processor data wherever possible to avoid sharing.
  - Segregate contended locks from their associated, frequently modified data. This keeps lock contenders from interfering with the lock holder.
  - Co-locate un-contended locks with their associated data to better utilize spatial locality and reduce the number of cache misses.

- Remote Memory Accesses
  - Ensure that read-mostly data is replicated into per-processor memory.
  - Migrate read/write data between per-processor memory if accessed primarily by one processor.
  - Replicate write-mostly data where possible and ensure replicas are in per-processor memory.

- Algorithmic
  - Use approximate local information rather than exact global information.
  - Avoid barriers

Replication, partitioning, migration and data placement are the key techniques advocated to implement these principles. Replication refers to the creation of local copies of data that can be locked and accessed locally. Partitioning is similar to replication but splits data into local components rather than making copies. Migration allows data to be moved to a location that provides greatest locality. Data placement refers to the use of padding and custom allocation routines to control where data is placed on cache lines and in the system's memory modules so as to avoid false sharing.

To illustrate the magnitude of the performance impact of contention and sharing, consider the following experiment: each processor continuously, in a tight loop, issues a request to a server, utilizing the Inter-Process Communication facility (IPC). The IPC from the client to the server and the request at the server are processed entirely on the same processor from which the request was issued, and no shared data needs to be accessed for the IPC. Using K42 on an S85 Enterprise Server IBM RS/6000 PowerPC bus-based cache-coherent multi-processor with 24 600MHZ RS64-IV processors and 16GB of main memory, the round trip IPC costs 1193 cycles. It involves an address

space switch, transfer of several arguments, and authentication on both the send and reply path.[2] The increment of a variable in the uncontended case adds a single cycle to this number. Figure 3.1 shows the performance of 4 variants of this experiment, measuring the number of cycles needed for each round-trip request-response transaction:

1. Increment a counter protected by a lock: each request is to increment a shared counter, where a lock is acquired before the increment. This variant is represented by the top-most curve: at 24 processors, each transaction is slowed down by approximately a factor of 19.

2. Increment a counter using an atomic increment operation. This variant shows a steady degradation where, at 24 processors, each request-response transaction is slowed down by a factor of about 12.

3. Increment a per-processor counter in an array. This variant has no logical sharing, but exhibits false sharing since multiple counters reside in a single cache line. In the worst case, each request-response transaction is slowed down by a factor of about 6. A knee appears at 16 processors due to the fact that 16 counters fit on a single cache-line on the machine used.

4. Increment a *padded* per-processor counter in an array. This variant is represented by the bottom-most curve that is entirely flat, indicating good speedup: up to 24 request-response transactions can be processed in parallel without interfering with each other.

These experiments show that any form of sharing in a critical path can be extremely costly — a simple mistake can cause one to quickly fall off of a performance cliff. Even though the potentially shared operation is, in the sequential case, less than one tenth of one percent of the execution time of the experiment, it quickly dominates performance if it is in a critical path on a multi-processor system. This kind of dramatic result strongly suggests that we must simplify the task of the developer as much as possible, providing abstractions and infrastructure that simplify the development of operating system code that minimizes sharing.

In general, data structures that may have been efficient in earlier systems, and might even possess high levels of concurrency, are often inappropriate in modern systems with high cache miss and write sharing costs. Moreover, as the example above demonstrates, a single poorly constructed component accessed in a critical path can have a serious performance impact. While the importance of locality has been recognized by many implementors of shared memory multi-processor operating

[2]The cost for an IPC to the kernel, where no context switch is required, is 721 cycles.

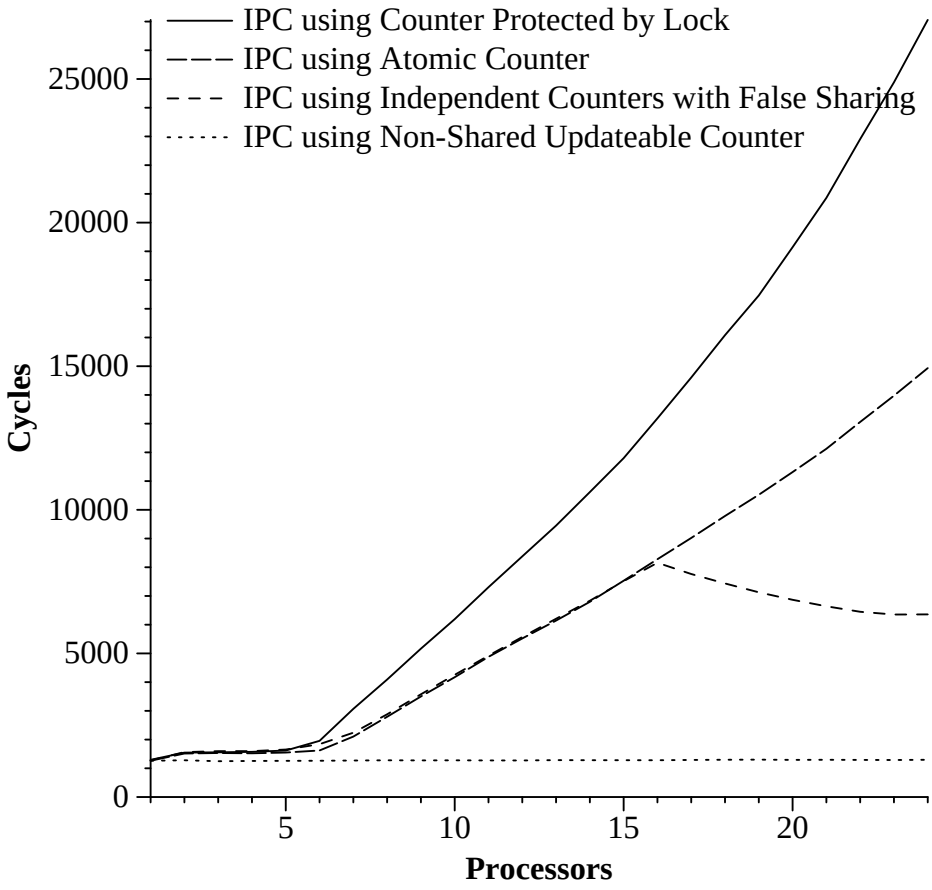


Figure 3.1: K42 microbenchmark measuring cycles required for parallel client-server IPC request to update counter. Locks, shared data access and falsely shared cache lines all independently increase round-trip times significantly. The knee at 16 processors, when using independent counters with false sharing, is due to the fact that 16 counters fit on a single L2 cache-line and at 17 processors the load is being distributed to two cache-lines.

systems, it can be extremely difficult to retrofit locality into existing operating system structures. The partitioning, distribution, and replication of data structures, as well as the algorithmic changes needed to improve locality, are difficult to isolate and implement in a modular fashion, given a traditional operating system structure.

The fact that existing operating system structures have performance problems, especially when supporting parallel applications, is exemplified in Figure 3.2, which shows the results of a few simple micro-benchmarks run on a number of commercial multi-processor operating systems[3]. For each commercial operating system considered, there is a significant slowdown when simple operations that should be serviceable completely independently of each other are issued in parallel. Micro-benchmarks are not necessarily a good measure of overall performance, however these results do

[3]It should be noted that these results are from 1999 and there have been advances in these commercial systems which are not reflected in these results.

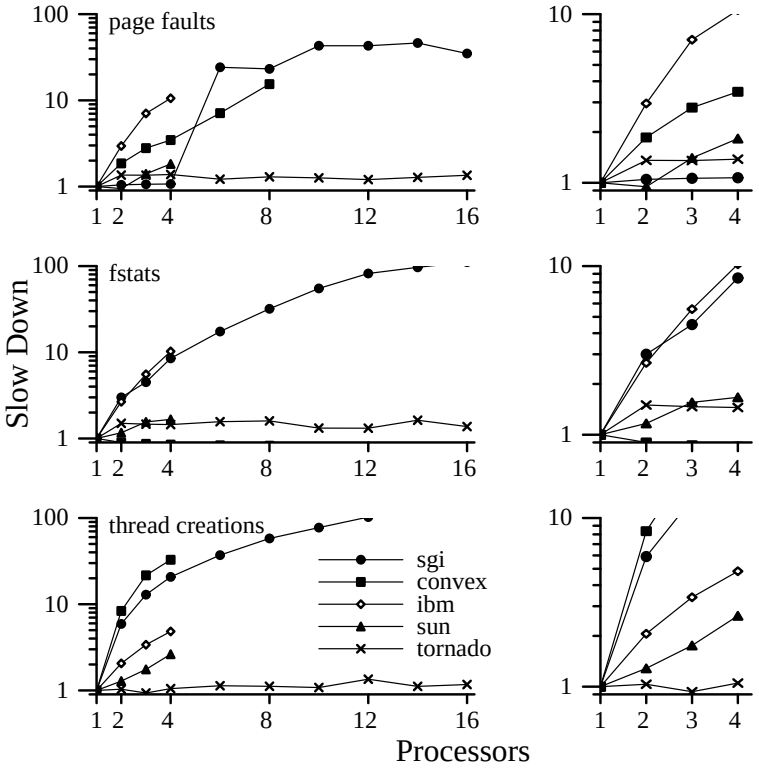


Figure 3.2: Normalized cost (log scale) of simultaneously performing on n processors: n in-core page faults (top), n fstats (middle), and n thread creations/deletions (bottom) for 5 commercial shared memory multiprocessor operating systems and for Tornado. The graphs on the right simply magnify the left-most part of the graphs on the left for clarity. A full description of these experiments is available elsewhere [49].

show that the existing systems can have performance problems.

3.2 Clustered Objects

One of the main contributions of this work has been to improve K42's scalability through the application of distribution to core system objects. The simple counter increment experiment demonstrates, that on a multi-processor, there is a large benefit to using a distributed counter rather than a single shared one. In the distributed case, each processor increments a per-processor sub-counter that does not require synchronization or shared memory access. When the value of the counter is required, the per-processor sub-counters are summed.

When implementing a distributed counter, there are a number of properties that are desirable. The distributed nature of the counter should be hidden behind its interface, preserving clean component boundaries and isolating client code from the distribution. An instance of the counter should have a single, unique, processor-independent identifier that transparently directs the caller

to the correct processor-specific sub-counter. Care must be used to ensure that the process for directing an access to a sub-counter does not require shared structures or incur significant costs in the common case. Using shared structures to access a per-processor sub-counter would defeat the purpose of trying to eliminate sharing. Further, if the costs to direct an access to the correct sub-counter are too expensive, then the use of this approach becomes questionable when the counter is only accessed on a single processor. To ensure scalability on systems with a large number of processors, a lazy approach to allocating the sub-counters is necessary. This ensures that the costs of allocating and initializing the sub-counter only occur on processors that access the counter.

To provide both uniprocessor and multi-processor components within a single object model, all K42's objects are implemented using Clustered Objects. Clustered Objects support distributed designs while preserving the benefits of a component-based approach. A Clustered Object can be internally decomposed into a group of cooperating subparts, called Representatives, that implement a uniform interface, but use distributed structures and algorithms to avoid shared memory and synchronization on its frequent and critical operations. Clustered Objects provide an infrastructure to implement both shared and distributed implementations of objects, and transparently to the client, permit the use of the implementation appropriate for the access pattern of the object. Collections of C++ classes are used to define a Clustered Object, and run-time mechanisms are used to support the dynamic aspects of the model.

In K42, we have re-implemented the low-level Clustered Object infrastructure (along with an extended garbage collection scheme) and implemented a new internal model of Clustered Objects and the associated protocols. Chapter 5 describes Clustered Objects and associated infrastructure and protocols in detail. The remainder of this section will provide an overview of Clustered Objects. The overview provides the background for the next chapter, which describes examples of the use of distribution in the construction and optimization of K42 system objects.

Clustered Objects allow each object instance to be decomposed into per-processor Representatives[4], and therefore provide a vehicle for distributed implementations of objects. Figure 3.3 abstractly illustrates a Clustered Object of a simple distributed integer counter. Externally, a single instance of the counter is visible, but internally, the implementation of the counter is distributed across a number of Representatives, each local to a processor. An invocation of a method of the Clustered Object's interface on a processor is automatically and transparently directed to the Rep-

[4]A Representative can be associated with a cluster of processors of an arbitrary size, from 1 to n, and not necessarily per processor.

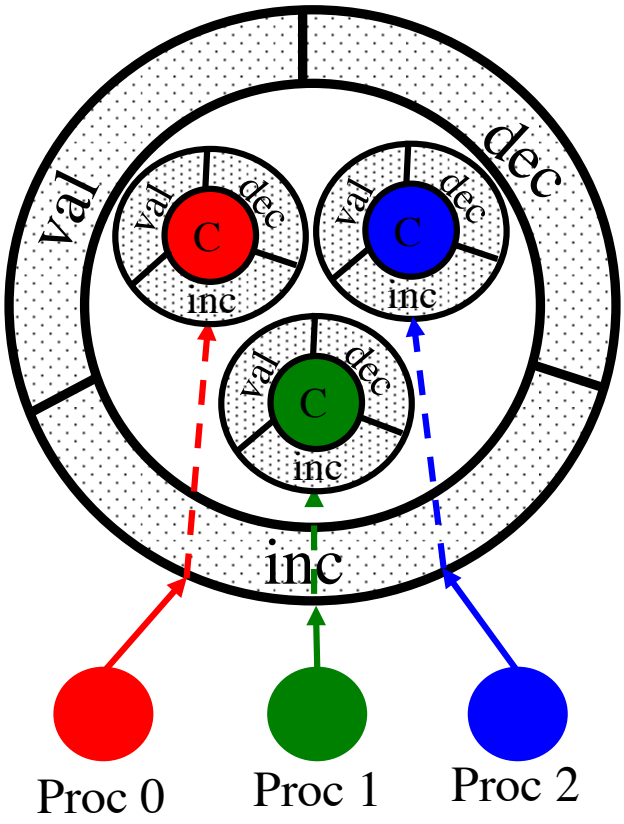


Figure 3.3: Abstract Clustered Object Distributed Counter

resentative local to the invoking processor. The internal distributed structure of a Clustered Object is encapsulated behind its interface and transparent to clients of the object. In Figure 3.3, a single instance of the counter, represented by the outer ring labeled with the counter's interface (`inc, val and dec`), is accessed by code executing on the processors at the bottom of the diagram. All processors invoke the `inc` method of the instance. Transparently to the invoking code, the invocations are directed to internal per-processor Representatives, illustrated by the three inner rings in the diagram. Each Representative supports the same interface but encapsulates its own data members. This ensures that the invocation of increment on each processor results in the update of an independent per-processor counter, avoiding sharing and ensuring good increment performance.

In the next chapter we will study our use of distribution in the construction of K42 system objects. The details of the Clustered Object infrastructure developed are presented in Chapter 5.

Chapter 4

Examples of Clustered Object Implementations

This chapter reviews the Clustered Objects we have designed and developed in K42 for distributing data structures in order to optimize performance and promote scalability, and presents associated experimental results.

4.1 Case Study: K42 VMM Objects

Virtual memory management (VMM) is one of the core services that a general purpose operating system provides and is typically both complex in function and critical to performance. Complexity is due primarily to i) the diversity of features and protocols that the virtual memory services must provide, and ii) the highly asynchronous and concurrent nature of VMM requests. The demand for performance is primarily on one VMM path – the resident page fault path. Generally, each executing program must establish access to the memory pages of its address space via page faults. Acceptable system performance is achieved by ensuring that most page faults can be satisfied without requiring IO by caching data in memory. Given temporal locality in page access, the page fault path to pages which are cached (resident) is performance critical (hot) with respect to address space establishment.

The optimization of the K42 VMM services has served as a case study for the application of distributed data structures to a core operating system service. As noted before, it is important that the services of an operating system be capable of reflecting the maximum concurrency of a workload. As a result, we have taken the opportunity to explore aggressive optimizations to enable

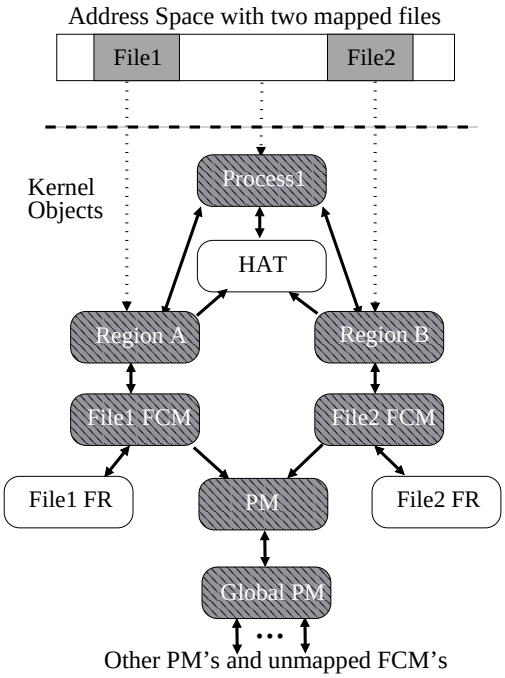


Figure 4.1: VMM Object Network representing an address space with two mapped files

concurrent faults to a single page without the need for synchronization and promoting locality in memory accesses in the common case. In this section, we review how distributed data structures encapsulated in the Clustered Objects of the K42 VMM are used to achieve the optimizations.

Figure 4.1 illustrates the basic K42 Kernel VMM Objects which are used to represent and manage the K42 virtual memory abstractions. Specifically, the figure illustrates the instances of objects and their interconnections that represent an address space with two independent files (File1 and File2) mapped in, as shown abstractly at the top of the diagram above the dashed line. The associated network of kernel objects is shown below the dashed line.

We have reimplemented each of the diagonally filled objects (Process, Region, File Cache Manager (*FCM*), Page Manager (*PM*) and Global Page Manager (*Global PM*)) in a distributed fashion and each is discussed in subsequent subsections. The other two objects, HATs, and FRs, are discussed in the following two paragraphs.

HAT The Hardware Address Translation (HAT) object is responsible for implementing the functions required for the manipulation of the hardware memory management units and is used to add a page mapping to the hardware-maintained translation services. In K42, a process's threads are scheduled on a per-process virtual processor abstraction which the OS maps to physical processors dynamically. As such, the page tables are managed on a per-virtual processor basis to allow

flexibility in K42's address space construction. To do this, the HAT object maintains page tables on a per-processor basis. Hence, the default HAT implementation is distributed by definition and naturally does not suffer contention. It thus requires no further distribution.

FR The File Representative (FR) objects are used to represent the file system entity of a particular file. The kernel uses the FR to communicate with the file system in order to conduct IO. There are two fundamental types of FRs: one that represents actual standard named files and one that represents anonymous files associated with computational mappings such as a process's heap. In the case of a named file, the FR is only accessed when IO operations are required (page faults to non-resident pages) and as such is heavy weight and infrequent. The FR implementation for a named file does use a single, shared, atomically updated counter, but, given that it is only accessed on a non-performance critical path, we have chosen not to distribute it. In the case of an anonymous file, the FR is responsible for determining if a page has been swapped to disk. As such, all faults are directed to the FR to determine whether this is an initial fault or whether the page has been accessed previously but is currently swapped out to secondary storage. If it is an initial fault, the FR returns an appropriate return code indicating that the page frame should be zero filled. Otherwise, it initiates IO to the swap device. Unless anonymous memory is swapped out, which indicates more serious memory pressure, the common case only requires the access of a read-mostly data variable on initial faults to an anonymous mapping. To date we have not observed or optimized performance of K42 under serious memory pressure and did not explore distributing FRs for anonymous mappings.

We now discuss the objects we distributed.

4.1.1 Process Object

The Process Object represents a running process and all per-process operations are directed to it. For example, every page fault incurred by a process is directed to its Process Object for handling.

The Process Object maintains address space mappings as a list of Region Objects. When a page fault occurs, it searches its list of Regions in order to direct the fault to the appropriate Region Object. The left-hand side of Figure 4.2 illustrates the default non-distributed implementation of the Process Object. A single linked list with an associated lock is used to maintain the Region List. To ensure correctness in the face of concurrent access, this lock is acquired on traversals and modifications of the list.

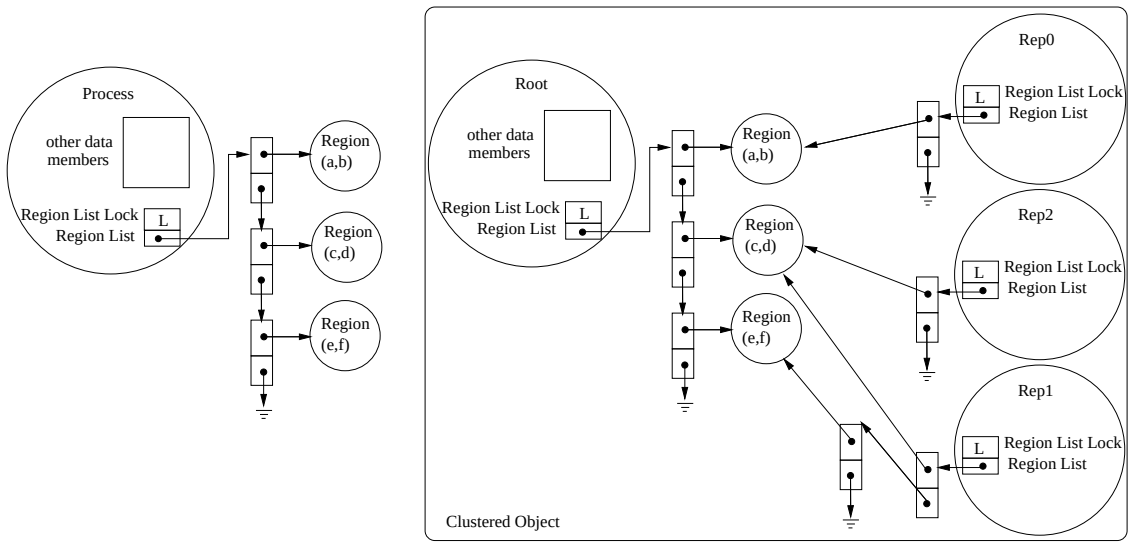


Figure 4.2: Non-Distributed and Distributed Process Objects

In the non-distributed implementation, the list and its associated lock can become a bottleneck in the face of concurrent faults. As the number of concurrent threads is increased, the likelihood of the lock being held grows, which can result in dramatic performance dropoffs as threads stall and enqueue on the lock. Even if the lock and data structures are not concurrently accessed, the read-write sharing of the cache line holding the lock and potential remote memory accesses for the region list elements can add significant overhead.

In order to gain insight into the performance implications of this implementation, let us consider a micro-benchmark which we will refer to as **memclone**. Memclone is modeled after a scientific application that utilizes one thread per processor to implement a distributed computation on a set of large arrays. Specifically, each thread allocates a large array which it then initializes sequentially in order to conduct its computation. This is modeled in the benchmark by having each thread sequentially access each virtual memory page associated with its array. This benchmark was sent to the IBM K42 group by one of IBM's Linux collaborators. The collaborators observed that the time to page fault the data pages of the arrays was limiting scalability on Linux. To understand the implications of K42's VMM structure on the performance of such a benchmark, we will briefly trace the behavior of K42's VMM objects during execution of the benchmark.

When each thread allocates its large array (on the order of 100 Megabytes) the standard Linux C library implementation of `malloc` will result in a new memory mapping being created for each array. In K42, this will result in a new Region and FCM being created for each array, where the FCM is attached to an FR which simply directs the FCM to fill newly accessed page frames with

zeros.

Thus, running the benchmark on n processors will result in the Process Object for the benchmark having n Regions, one per array, added to its region list. As each thread of the benchmark sequentially accesses the virtual memory pages of its array, a page fault will occur for each page accessed. This will result in a page fault exception occurring on the processor on which the thread is executing.

K42's exception level will direct the fault to the Process Object associated with the benchmark, by invoking a method of the Process Object. The Process Object will search the region list to find the Region responsible for the address of the page which suffered the page fault and then invoke a method of the Region. The Region will translate the faulting address into a file offset and then invoke a method of the FCM it is mapping. The FCM will search its data structures to determine whether the associated file page is already present in memory. In the case of the page faults induced by the benchmark, simulating initialization of the data arrays, all faults will be for pages which are not present. As such, the FCM will not find an associated page frame and will allocate a new page frame by invoking a method of the PM (Page Manager) to which it is connected. The PM will then invoke a method of the Global PM to allocate the page frame. To complete the fault, the FCM initializes the new page frame with zeros, and then maps the page into the address via the HAT object passed to it from the Process.

When considering K42's VMM structure, we observe that the memclone benchmark will result in each thread accessing independent Regions and independent associated FCM's. Therefore, we expect these objects to be accessed exclusively by separate processors, and thus suffer no contention and have good locality. The HAT, PMs and Process objects, however, are commonly accessed by all page faults and thus may suffer contention-inducing remote accesses. As discussed earlier, the HAT is not of concern, as, in the common case, it only accesses local processor state in order to establish the page mapping by manipulating a processor's local page table. The PM associated with the Process Object does not do significant work on the page allocation path other than directing the request to the Global PM[1]. The only two objects that may suffer real contention are the Process and Global PM objects. In order to isolate the impact of the Process Object on performance we will assume that the optimized version of the Global PM is being used (discussed in Section 4.1.2).

The distributed Process Object is designed to cache the region list elements in a per-processor

[1]The primary role for the PM attached to the process is to record the FCMs backing the computation/heap regions of a process for the sake of paging.

Representative (see right-hand side of Figure 4.2). A master list, identical to the list maintained in the non-distributed version, is maintained in the root. When a fault occurs, the cache of the region list in the local Representative is first consulted, acquiring only the local lock for uniprocessor correctness. If the region is not found there, the master list in the root is consulted and the result is cached in the local list, acquiring and releasing the locks appropriately to ensure the required atomicity. This approach ensures that in general, the most common operation (looking up a region) will only access memory local to the processor and not require any inter-processor communication or synchronization.

It should be noted that the distributed object has greater costs associated with region attachment and removal than the non-distributed implementation. When a new region is mapped, the new Region Object is first added to the master list in the root and then cached in the local list of the processor mapping the region. To un-map a region, its Region Object must atomically be found and removed first from the root and then from all Representatives that are caching the mapping. A lookup for a region not present in the local cache requires multiple searches and additional work to establish the mapping in the local list. As in the case of a multi-threaded process such as memclone, the overhead of the distributed implementation for region attachment and removal is more than made up for by the more efficient and frequent lookup operations.

In the case of a single-threaded application, all faults occur on a single processor and the distributed version provides no benefit, resulting in additional overheads both in terms of space and time. In order to maximize performance we advocate that the hot-swapping mechanisms, discussed in Chapter 6, be used so that when a process is created, the non-distributed Process Object is used by default. The distributed implementation should be automatically switched to when a process becomes multi-threaded, using the hot-swapping facility.

Although one can imagine more complex schemes for implementing distributed versions of the Process Object, the simple scheme chosen has three main advantages:

1. Its performance characteristics are clear and easy to understand.

2. It preserves the majority of function of the non-distributed version, and also making the implementation easy to understand and maintain.

3. The distributed logic integrates easily into the non-distributed behavior, relying on a simple model of caching which is implemented in a straightforward manner on top of the pre-existing data structures and facilities: a simple list which uses a single lock (identical to the list used

in the non-distributed version), and the standard Clustered Object infrastructure for locating data members of the root object and the ability to iterate over the Representatives.

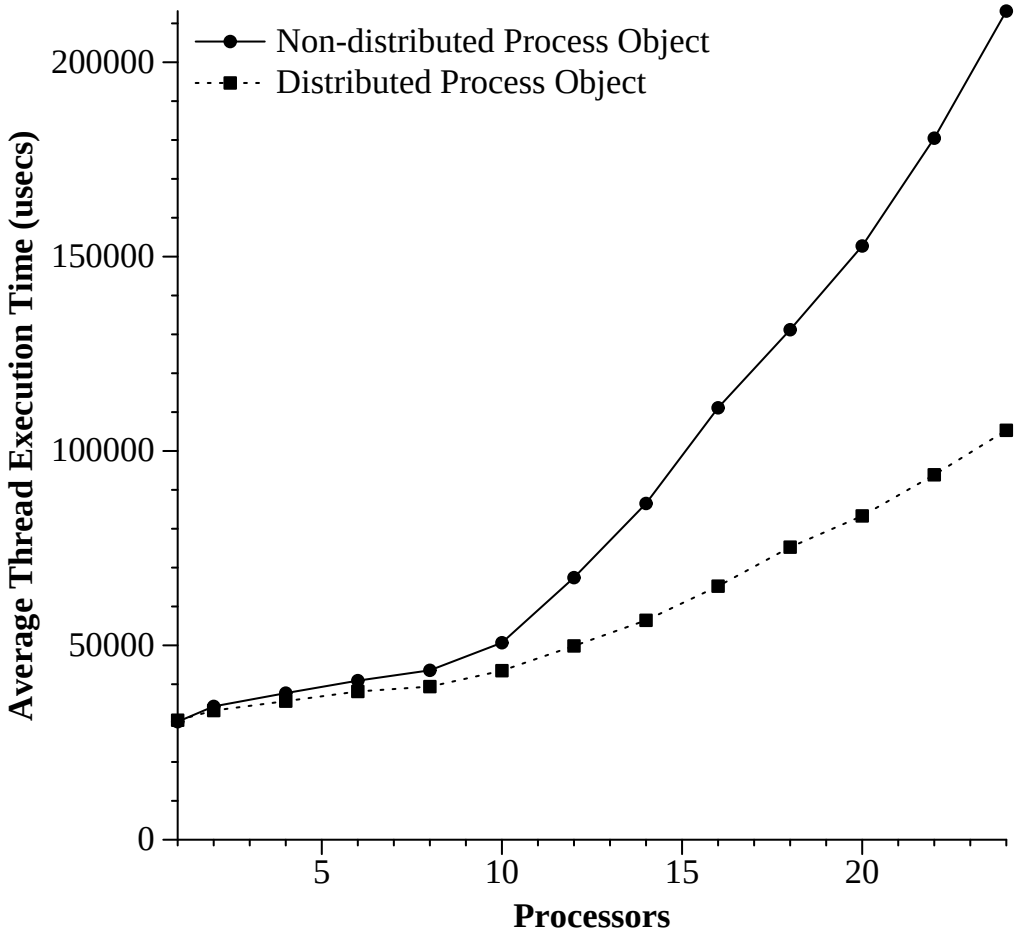


Figure 4.3: Graph of average thread execution time for memclone running on K42. One thread is executed per-processor. Each thread touches 2500 pages of an independent memory allocation. Each page touch results in a zero-fill page fault in the OS. The performance of using non-distributed and distributed Process Object is shown (in both cases the distributed Global PM Object is used). Ideal scalability would be represented by a horizontal line.

Figure 4.3 displays the performance of memclone running on K42 with a non-distributed version and the distributed version of the Process Object. We have instrumented memclone to measure the execution time of each thread. Each thread of memclone touches 2500 pages of memory and one thread is executed per-processor. The average execution time is measured and plotted in the graph. Ideal scalability would correspond to a horizontal line on the graph. We see that beyond 10 processors, the performance of the non-distributed version degrades rapidly. The distributed Process Object does perform better, however its scalability is still not ideal. Our tools do not indicate any contention in a particular object or show a particular path as unduly dominating

performance. Using a test which concurrently exercises the page fault exception and page mapping features of each processor, without access to the VMM data structures of the OS, reveals a slow down which we believe is particular to the S85 Enterprise Server IBM RS/6000, on which our experiments are run. Without more advanced hardware and software tools however, it is not feasible to isolate the exact source of the slowdown.

4.1.2 Page Managers and the Global Page Manager

The Page Manager (PM) object manages and tracks the physical page frames associated with a process and is responsible for satisfying requests for page frames as well as reclaiming pages when requested to do so. The PMs are organized as a hierarchy, in which the PM associated with each process is attached to a Global PM which globally manages physical page frames of the system. In K42, the implementation of the PMs associated with each process are trivial and do little beyond redirecting calls to the Global PM[2]. The only real service that the initial PM implementation provides is to track the list of File Cache Management Objects (FCM's) created by a process for its computational regions. For completeness we have distributed this list of FCM's in a manner very similar to that of the Global Page Manager described in detail below. In the long run, however, we expect the PM instances associated with each process to serve a more fundamental role in implementing a per-process working set model of physical memory management and we expect to have to reimplement it in the future as necessary.

The Global Page Manager, the object at the root of the hierarchical configuration of co-operating Page Managers (PMs), is responsible for physical page frame management across all address spaces in the system[3]. All memory regions of an address space in K42 are mapped by Region Objects to a specific file via a File Cache Management Object (FCM). This includes explicitly named files such as a program executable, as well as the anonymous files associated with computational regions such as a process heap. FCM's maintain the resident pages of a file and are attached to a Page Manager Object, from which they allocate and deallocate physical pages. FCM's for named files opened by processes are attached to the Global Page Manager. The Global Page Manager implements reclamation by requesting pages back from the FCM's attached to it and from the Page Managers below it. Each Page Manager below the Global Page Manager is attached to a Process Object and implements page management for the FCM's associated with computational regions of the process.

[2]The main purpose of the PMs associated with each process is to provide a degree of freedom for future exploration of more complex VMM implementations.

[3]K42 employs a working set page management strategy.

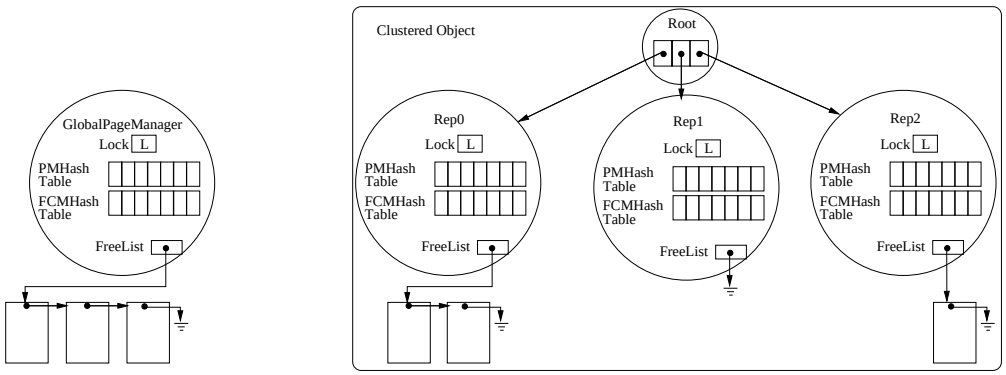


Figure 4.4: Non-Distributed vs Distributed Global Page Manager

The left-hand side of Figure 4.4 illustrates the simple non-distributed implementation of the Global Page Manager that was first used in K42. It contains a free list of physical pages and two hash tables to record the attached FCM's and PMs. All three data structures are protected by a single, shared lock. On allocation and deallocation requests, the lock is acquired and the free list manipulated. Similarly, when a PM or FCM is attached or removed, the lock is acquired and the appropriate hash table updated. Reclamation is implemented as locked iterations over the FCM's and PMs in the hash tables with each FCM and PM instructed to give back pages during the reclamation iterations.

As the system matured, we progressively distributed the Global Page Manager Object in order to alleviate the contention observed on the single lock. The most recent distributed implementation is illustrated on the right-hand side of Figure 4.4. The first change was to introduce multiple Representatives and maintain the free lists on a per-processor basis. The next change was to partition the FCM Hash Table on a per-processor basis by placing a separate FCM Hash Table into each Representative and efficiently mapping each FCM to a particular Representative.

In the distributed version, page allocations and deallocations are done on a per-processor basis by consulting only the per-Representative free list in the common case, which improves scalability. The current implementation uses a very primitive scheme to balance the free lists across Representatives. A global free/overflow list is placed in the root of the distributed implementation. Initially, the Global PM seeds its free memory via requests to a physical frame allocation service of K42, the description of which is beyond the scope of this thesis. In steady state the free pageable memory of the system resides on the free lists of the Global PM. When the free list of a Representative exceeds a preset threshold, the Representative moves a pre-defined number of pages onto the overflow list maintained in the root. When a Representative cannot satisfy a request, the Representative

attempts to replenish its free list by moving a pre-defined number of pages from the overflow list. If the overflow is empty, a new chunk of pages is requested from the pool of un-allocated physical memory. In the extreme, if no physical memory is available, the Global PM will initiate its paging algorithms in order to reclaim currently inactive pages. The current approach is preliminary in nature and by no means complete, relying on a number of hard coded constants and unverified heuristics. However, a complete study of these policies has not yet proven necessary. The encapsulation afforded by the Clustered Object approach has allowed us to incrementally improve and study the behaviour of the Global PM.

The mapping of FCM's to an appropriate Global PM Representative is illustrative of the distributed techniques exploited to avoid contention. To provide efficient Clustered Object allocation, Clustered Object identifiers are allocated on a per-processor basis such that a Clustered Object allocated on a given processor is assigned an identifier from a range specific to that processor. Conversely, given a Clustered Object identifier, the processor on which it was allocated can be efficiently determined. This calculation has been implemented as a general service of the Clustered Object manager. In the case of the distributed Global Page Manager, the allocating processor for an FCM is treated as its home processor. FCM attachment and detachment to the Global Page Manager is achieved by invoking the service of the Clustered Object Manager to determine the allocating processor for the FCM being attached or detached. Then the FCM is recorded in the hash table of the Global Page Manager Representative associated with that processor. Given that there is only one instance of the Global Page Manager in the system and that there is a Representative for every processor, an array of Representative pointers was put into the root of the distributed Global Page Manager to facilitate more efficient mapping of processor to Representative. The Clustered Object class infrastructure developed provides facilities for locating a Representative, given a processor number, but it is designed for general use and is thus more costly. A similar approach is used to implement PM attachment and detachment.

Currently the Global PM implements a very simple paging algorithm, where page reclamation is done as an iteration over the Representatives of the Global Page Manager. Each Representative iterates over the FCM's recorded in its hash tables, querying each for pages. The Global Page Manager then iterates over the PMs recorded in its PM hash table. In the long run, we expect to parallelize this algorithm more aggressively with multiple concurrent threads implementing reclamation on a per-processor basis, utilizing the distributed Representative structure of the Global Page Manager.

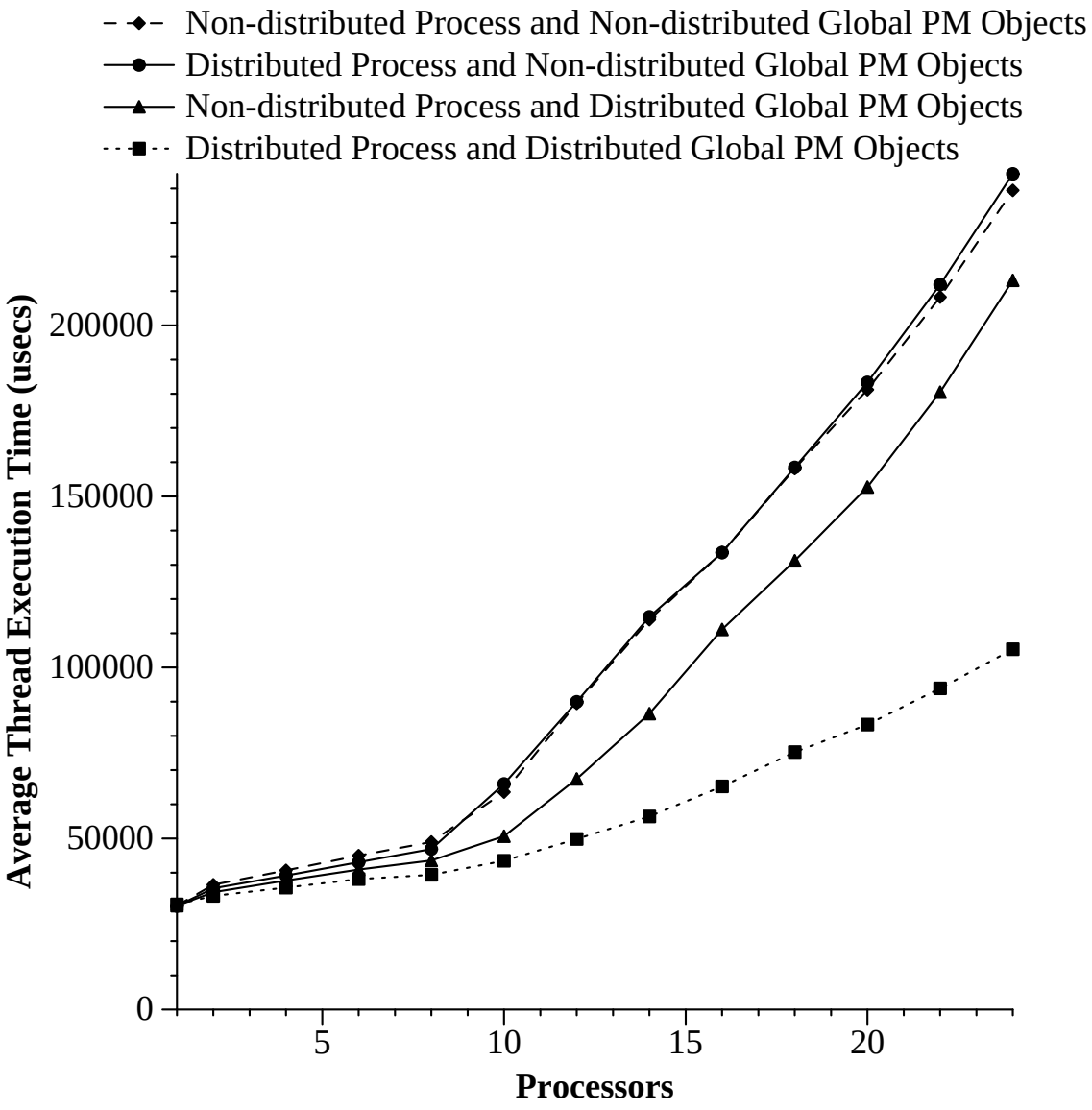


Figure 4.5: Graph of average thread execution time for memclone running on K42. One thread is executed per-processor. Each thread touches 2500 pages of an independent memory allocation. Each page touch results in a zero-fill page fault in the OS. The performance of using all four combinations of shared and distributed implementations of the Process and Global PM objects are shown. Ideal scalability would be represented by a horizontal line.

In Figure 4.5, we plot the performance of an instrumented version of memclone, measuring the average execution time of the threads using the various combinations of Process and Global PM implementations. Each thread, again, touches 2500 pages of memory and one thread is executed per-processor. Ideal scalability would be a horizontal line on the graph. Focusing on the two curves labeled *Distributed Process and Non-distributed Global PM Objects* and *Distributed Process and Distributed Global PM Objects* we can consider the performance impact of the Global PM implementations. Beyond 8 processors a non-distributed version results in a sharp decrease in performance. The distributed version yields better scalability however, a gradual decrease in performance is also observed. As discussed in the previous subsection, we believe characteristics of the S85 hardware are limiting scalability when using our distributed objects for this micro-benchmark.

When we consider all curves of Figure 4.5, it is clear that using only one of the distributed implementations is not sufficient. The use of both distributed implementations of the Process Objects and the Global PM Object are required in order to obtain a significant improvement in scalability. This is indicative of our general multi-processor optimization experience, where optimizations in isolation often affect performance in non-intuitive ways, such that the importance of one optimization over another is not obvious. A key benefit of the Clustered Object approach is that every object in the system is eligible for optimization, not just the components or code paths that the developer might have identified a priori.

4.1.3 Region

The Region Object is responsible for representing the mapping of a portion of a file to a portion of a process's address space. It serves two purposes. Firstly, it translates the address of a page fault to an appropriate file offset which is being mapped by that portion of the address space. Secondly, it provides synchronization between various operations to the mapping, so that the following properties hold:

1. Construction of the mapping (the associated objects and interconnections) does not conflict with requests to un-map it or access it via page faults.

2. When an un-map request has been made, current in-flight page faults are allowed to complete prior to destroying the objects that represent the mapping.

3. When an un-map request is made, new page faults to that portion of the address space are rejected, while the un-mapping operation is being conducted.

4. Only one un-map request is executed and all others made before the first is completed are rejected.

5. When an un-map request is made, no new mappings can be established which overlap with the original, until the un-map operation has completed.

6. Operations on a mapping are suspended while its bounds are changed, e.g., the region is being truncated.

7. Operations on a mapping are suspended while a copy of the mapping is being created due to a fork operation (e.g., creates a copy of itself).

To ensure the above properties, the original version of the Region Object was implemented using a form of reference counting to provide the necessary synchronization. A single request counter abstraction in the Region object is used to record all relevant requests (method invocations). At the start of a request an `enter` operation is performed and on exit a `leave` operation is performed on the counter. Depending on the state of the Region the `enter` operation will: 1) atomically increment the counter, or 2) fail indicating that the request should be rejected, or 3) internally block the request. The behaviour of the request counter is controlled by a control interface. Ignoring the details, the request counter is fundamentally a single shared integer variable which is atomically incremented, decremented and tested on all relevant methods of the Region object, including the `handleFault` method which is on the resident page fault path. [4]

If multiple threads of an application are faulting on the same portion of their address space, we expect that the request counter in the Region Object will suffer contention. To understand how parallel faults to a common portion of the address might occur, consider a modified version of the memclone benchmark, in which a common data array is allocated and each thread accesses a portion of the common array rather than independent arrays. A common array can be allocated either by explicitly creating a common mapping (e.g., an mmap of /dev/zero) or by a single large heap allocation (e.g., a single large malloc). This would result in the page faults induced by all threads of memclone being serviced by a common Region and FCM, potentially causing contention (with the design of the FCM being discussed in the next subsection).

Beyond the synchronization provided by the Region object, the Region Object is very simple, requiring a small number of scalar variables which are, in the common case, read only with respect

[4] In the long run we expect to replace the request count based Region implementation with a Read-Copy-Update implementation, once a generic service such as QDO, briefly discussed in Chapter 8, is implemented.

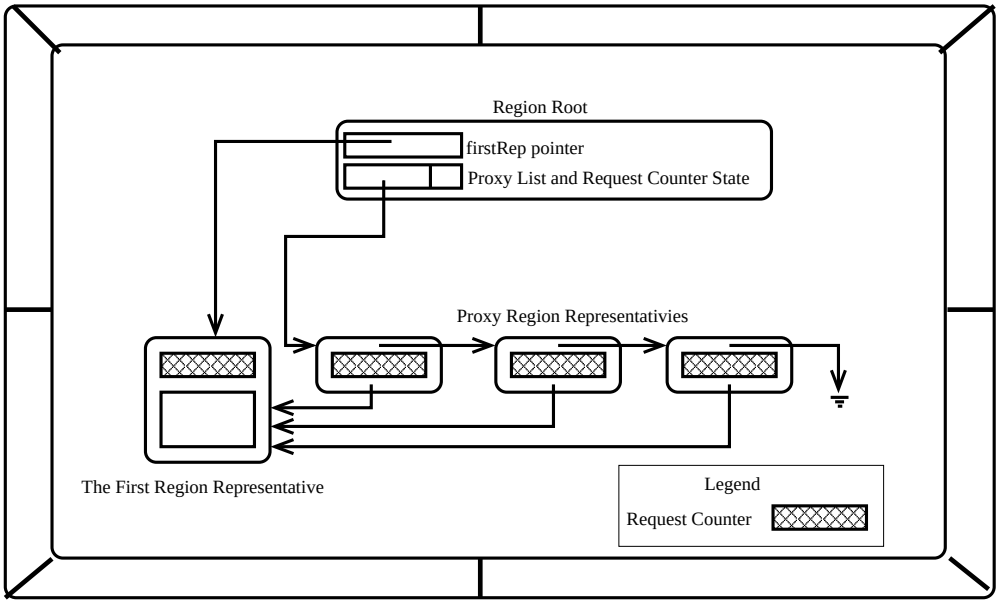


Figure 4.6: Distributed Region Clustered Object. There is a single "real" firstRep Representative. All other Representatives are proxies which only contain a request counter. Most calls to the proxy are simply forwarded to the firstRep, however the handleFault method manipulates the request counter of the proxy and then forwards its call to a method of the firstRep which implements the main logic of the call. Thus, with respect to the fault handling path the only read write variable is manipulated locally. The root maintains the list of proxies as well as the state of the request counters globally in a single integral variable which can be updated atomically.

to the `handleFault` method. To this end, we have taken an approach which just focuses on distributing the request count to alleviate contention it may induce on the resident page fault path (as illustrated in Figure 4.6). We have constructed a version which creates a single Representative (the *firstRep*) when the distributed Region object is created. This Representative is nearly identical to the original shared version except that the Root for the distributed version is constructed to create specialized Representatives, which we refer to as proxy Representatives, for any additional processor that may access the Region instance. For all methods other than the crucial `handleFault` method, the proxy Representatives simply invoke the equivalent method on the *firstRep* since we don't expect contention on the other operations. In the case of the `handleFault` method, which is the only method that invokes the `enter` of the request counter, the proxy Representative version does an `enter` operation on a request counter local to it and then invokes the body of the standard `handleFault` method on the *firstRep*. Figure 4.7 shows a fragment of the proxy Representative class of the distributed region. The main point to note is that each proxy instance has its own request counter. The classes of the distributed region cooperate to ensure that the local request counters are globally consistent to ensure correctness on all paths in a manner that allows the

`handleFault` method to only require manipulation of the local request counter.

A fragment of the class for the *firstRep* is shown in Figure 4.8. The majority of its implementation is inherited from the original non-distributed version of the Region. Only the methods relevant to the manipulation of the request counter are overridden to account for the use of the multiple request counts. The main point to note is that for an instance of the distributed Region there is one *firstRep* Representative which serves as the main coordination point for the distributed request counters. Its request counter is always manipulated first with respect to the control interface (e.g., stop, restart, and shutdown). This ensures that the distributed version of these preserves the semantics of a single request counter. For example, if two `stops` occur on different processors, the second (temporally) will fail as expected since both will first attempt the `stop` operation on the request counter of the `firstRep`. However, if they were to have attempted to manipulate an arbitrary local request counter both may have succeeded leading to potentially un-deterministic behaviour. As such, the control methods manipulate the local request counter of *firstRep* and then invoke routines of the distributed Region's root to change the global state of the request counters and manipulate the request counts of any proxies in a consistent manner.

The majority of the complexity and new functionality associated with the distributed Region has been factored into the root class. Figures 4.9 and 4.10 illustrate fragments of the distributed Region root class. The root's methods are constructed to ensure correct behaviour in the face of dynamic additions of Representatives. As is standard for distributed Clustered Objects, new accesses to the Region on other processors can require the creation of new Representatives via a standard instantiation protocol (Clustered Object miss-handling protocol) discussed in detail in the next chapter. The protocol requires that a developer specify a `createRep` method in the root of the Clustered Object which creates and initializes a new Representative for the processor which accessed the object. In the case of the distributed Region's root, this method creates a proxy Representative and then initializes the state of the request counter associated with the new Representative. The next two paragraphs briefly discuss how the Root of the distributed region synchronizes Representative creation with concurrent changes to the request counters' state.

As illustrated by the `stopOtherRequests` method, the methods that manipulate the state of the request counter are coded to adhere to synchronization rules which ensure that changing the state of the request counters and addition of proxies occurs atomically. To achieve this, a single atomically manipulable data value, `currentState`, is used, which encodes the global state of all

```
#define REDIRECTTO(m) firstRep->m
class ProxyDistRegion : public Region {
    DistRegionDefault *firstRep;
    ProxyDistRegion   *nxt;
    RequestCountWithStop requests;
    ProxyDistRegion(DistRegionDefault *fRep, ProxyDistRegion *n) :
        firstRep(fRep), nxt(n) { }

    ProxyDistRegion * getNext() { return nxt; }
    sval enterRequest()         { return requests.enter(); }
    void leaveRequest()         { requests.leave(); }
    sval stopRequests()         { return requests.stop(); }
    void restartRequests()      { requests.restart(); }
    sval shutdownRequests()     { return requests.shutdown(); }

public:
    virtual SysStatusUval handleFault(AccessMode::pageFaultInfo pfinfo,
                                      uval vaddr,
                                      PageFaultNotification *pn,
                                      VPNum vp)
        {
            SysStatusUval rc;
            if (enterRequest() < 0) {
                //N.B. when enter fails, DO NOT leave()
                return _SERROR(1214, 0, EFAULT);
            }

            rc=REDIRECTTO(requestCounted_handleFault(pfinfo,vaddr,
                                                     pn,vp));

            leaveRequest();
            return (rc);
        }

    virtual SysStatus forkCopy(FRRef& fr)
        { return REDIRECTTO(forkCopy(fr)); }
    .
    .
    .
};
```

Figure 4.7: Code fragment from the proxy Representative of the distributed Region object. The single relevant request counter data member `requests` as well as the `handleFault` method is shown. All other methods, such as `forkCopy`, are unconditionally redirected to the *firstRep*. Redirection is achieved by the trivial `REDIRECTTO` macro which simply dereferences a pointer to the *firstRep* and invokes the specified method.

```
public:
    sval enterRequest()     { return requests.enter(); }
    void leaveRequest()     { requests.leave(); }

    // May want to use method pointers for these (or real hot swapping :-))
    sval stopRequests()
        {
            rc=requestCount.stop();
            if (rc < 0) return rc;
            return COGLOBAL(stopOtherRequests(rc));
        }
    void restartRequests()
        {
            COGLOBAL(restartOtherRequests());
            requests.restart();
        }
    sval shutdownRequests()
        {
           rc=requests.shutdown();
           if (rc < 0) return rc;
           return COGLOBAL(shutdownOtherRequests(rc));
        }
```

Figure 4.8: Code fragment of some of the distributed Region `firstRep` implementation. The the control operations of the request counter have been modified so that each operator now first manipulates the request counter associated with the `firstRep`. The operations then invoke a method of the new root for the distributed counter to manipulate the request counters of the proxy Representatives that may exist.

```
class DistRegionDefaultRoot : public CObjRootMultiRep {

    struct State : BitStructure {
        __BIT_FIELD(62, proxyListHeadBits, BIT_FIELD_START);
        __BIT_FIELD(2, rcState, proxyListHeadBits);
        enum RCStates {NORMAL=0, STOP=1, SHUTDOWN=2};

        ProxyDistRegion *proxyListHead()
            {
                // to turn bits back into a pointer must shift them up 2
                return (Proxy *)(proxyListHeadBits()<<2);
            }

        void  proxyListHead(ProxyDistRegion *head)
            {
                // chop off bottom two bits from pointer
                proxyListHeadBits(uval64(head)>>2);
            }

        } currentState;

    DistRegionDefault *firstRep;
```

Figure 4.9: Code fragment showing the key data members of the root class created for the distributed Region object. The critical member is `currentState` which is used to encode both the global state of the request counter as well as the list of Representatives.

```
    virtual CObjRep * createRep(VPNum vp)
        {
           ProxyDistRegion *proxy=0;
           State oldState,newState;
           // Atomically add a proxy
           do {
                oldState = currentState;
                newState = oldState;
                if (proxy) delete proxy;
                proxy=new ProxyDistRegion(firstRep,newState.proxyListHead());
                if (newState.rcState()==State::STOP) {
                   proxy->requests.stop();
                } else if (newState.rcState()==State::SHUTDOWN) {
                   proxy->requests.shutdown();
                }
                newState.proxyListHead(proxy);
           } while (!CompareAndStore64Synced(&currentState.data,
                                             oldState.data,
                                             newState.data));
           return (CObjRep *)proxy;
        }

    sval stopOtherRequests(sval rc)
        {
           State oldState, newState;
           sval rtn=0;
           // update request counter state.
           // doing so now ensures that any new
           // proxys added will be in the right state
           do {
                oldState = currentState;
                newState = oldState;
                newState.rcState(State::STOP);
           } while (!CompareAndStore64Synced(&currentState.data,
                                             oldState.data,
                                             newState.data));
           // now update the request counters of the proxys
           // that exist to match
           for (ProxyDistRegion *p=newState.proxyListHead();
                p!=0;
                p=p->getNext()) {
               rtn+=p->stopRequests();
               tassertMsg(rtn==0, "oops proxy already stopped\n");
           }
           return rtn;
        }
};
```

Figure 4.10: Code fragment of the root class created for the distributed Region object. The `createRep` method, invoked when a new Representative is required, creates proxy Representatives. It initializes the proxy Representatives in an atomic fashion, so that their request counters are consistent with the current global view. Similarly the control operations of the request count must be appropriately synchronized. The `stopOtherRequests` is shown as an example.

the proxies as well as the pointer to the head of the proxy list[5]. Thus, carefully coded loops, as illustrated in the `createRep` method and `stopOtherRequests`, can be used to ensure that changes of state and addition of proxies are mutually exclusive. Such loops use standard non-blocking techniques in which copies of the current state are used to construct the intended change followed by an atomic compare and swap attempt to commit the changes. If the compare and swap fails then the change was not successful and must be retried based on the new state.

Furthermore, more subtle issues regarding deadlock are avoided by the implementation. The standard Clustered Object classes provide a `repLock` and a list of all Representatives. The lock is used to ensure mutual exclusion for the creation of Representatives and for simple traversals of all the Representatives, such as scatter or gather operations. The `repLock` can be acquired and the standard list of Representatives can be used to iterate over the Representatives. Doing so ensures that the list of Representatives stays stable by suspending any attempts to initiate the Clustered Object miss-handling protocol. In general, if the operation that is being carried out over the Representatives only requires manipulation of the data members of the Representative then such an approach of using the `repLock` and standard list of Representatives does not pose a problem.

If however, the operations while holding the `repLock` directly or indirectly invoke other objects, then it is possible that an attempt to access the original Clustered Object via its Clustered Object reference may occur. This may cause the initiation of the Clustered Object miss-handling protocol on the Clustered Object for which the `repLock` is held, resulting in a deadlock. For example, if the `repLock` and standard Representative list were used by the distributed Region to implement the loop to invoke the stop operation on each Representative's request count, a deadlock could occur. While holding the `repLock`, the thread iterating over the Representatives may block when it invokes the stop operations in order to wait for concurrent accesses by other threads, which had already incremented the request count, to complete. However, the threads executing those accesses may cause an access to the object via its Clustered Object identifier causing the miss-handling protocol to be initiated. But since the `repLock` is held by the thread which is waiting for the request count to reach zero, the thread executing the miss-handling protocol will block resulting in a deadlock.

There is a software engineering advantage to constructing the distributed Region Object using proxies. Adapting the non-distributed implementation with proxies to construct the distributed

[5]The design requires access to `currentState` on the infrequent operations which change the global behaviour of the counters and when a new representative is created

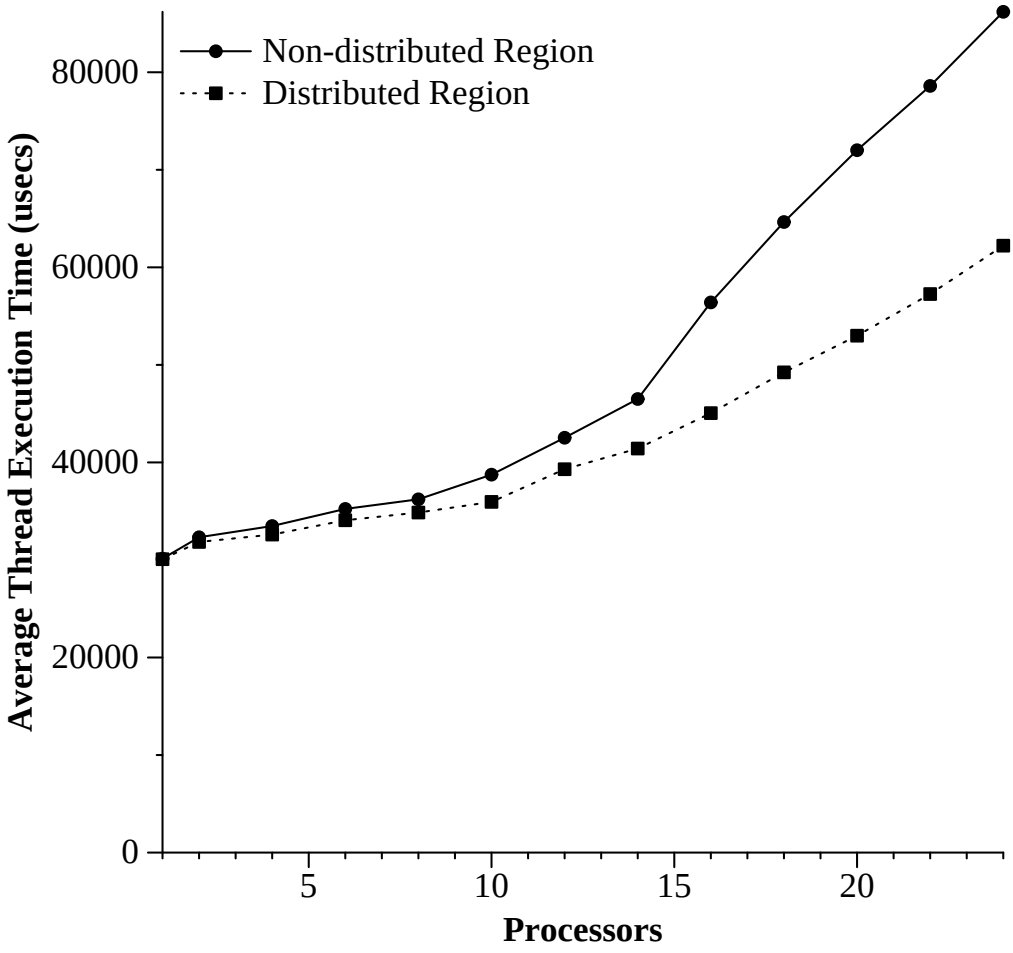


Figure 4.11: Graph of average thread execution time for a modified memclone running on K42. One thread is executed per-processor. Each thread touches 2500 independent pages of a single large common memory allocation. Each page touch results in a zero-fill page fault in the OS. The graph plots the performance of using a non-distributed versus a distributed Region Object to back the single common memory allocation. Ideal scalability would be represented by a horizontal line.

implementation results in the main logic of the object continuing to reside in the single C++ class of the non-distributed Representative. This avoids having to create a duplicate independent distributed implementation, thus avoiding the need to maintain two implementations as the system evolves. Unfortunately this approach cannot always be used, as the more complex the object is, the less likely a distributed version can be constructed from the shared implementation.

Figure 4.11 displays a graph of the performance obtained when running a modified version of memclone on K42 using the non-distributed versus the distributed Region object. In the modified version of memclone, a single large memory allocation is used for a common array, which is accessed in a partitioned fashion. One thread is executed per-processor and each touches 2500 pages unique to the thread, however the pages are from a common array created via a single allocation. The zero-

fill page faults that the threads induce will therefore be serviced not only by a common Process Object and the Global PM but also by a common Region and FCM which represent the single memory allocation. In the performance data presented in the figure, all objects other than the Region backing the allocation are using distributed implementations so that we can focus on the impact of the Region in isolation. Again, ideal scalability would be manifested as a horizontal line on the graph. Although the difference between the non-distributed and distributed Region is not as dramatic as the effects of distributing the Process and PM objects, it is worth noting that even a simple object, which is not much more than a counter, has an effect on scalability. As discussed previously, we believe characteristics of the S85 hardware are limiting scalability when using our distributed objects for this micro-benchmark.

4.1.4 File Cache Manager

All regions of an address space are attached to an instance of a File Cache Manager (FCM), which caches the pages for that region. (Recall that an FCM may be backed by a named file or swap space in the case of anonymous regions.) An FCM is responsible for all aspects of resident page management for the file it caches. On a page fault, a Region Object asks the FCM to which it is attached to translate a file offset to a physical page frame. The translation may involve the allocation of new physical pages and the initiation of requests to a file system for the data. With respect to paging, when a Page Manager asks it to give back pages, an FCM must implement local page reclamation over the pages it caches. The FCM is a complex object, implementing a number of intricate synchronization protocols including:

1. race-free page mapping and un-mapping,
2. asynchronous I/O between the faulting process and the file system,
3. timely and efficient page reclamation, and
4. maintenance of fork logic in the case of anonymous regions.

The standard, non-distributed FCM uses a single lock to ease the complexity of its internal implementation. When a file is accessed by a single process, the lock and centralized data structures do not pose a problem. When many processes or threads of a single process access the file concurrently, however, then the shared lock and data structures induce inter-processor communication resulting in degraded page fault performance.

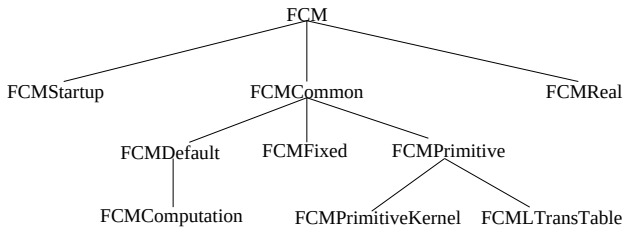


Figure 4.12: Original FCM Class Hierarchy

This section is presented in a style which documents not only what solutions we have derived to address the challenges of distributing the FCM but also the process for arriving at them. The construction and optimization of complex systems software components, in a large pre-existing operating system source base is a challenging task. Our experience illustrates both the complexity of developing a distributed implementation as well as the practical challenges a developer faces in introducing such an implementation into K42. Clustered Objects help address some of the complexity, but by no means makes the task of introducing distribution trivial.

Unlike the Process, Global Page Manager and Region objects, there is no straightforward way to distribute the FCM's data without adding considerable complexity and breaking internal protocols. Figure 4.12 illustrates the original FCM class hierarchy. At the top is the FCM class which defines the basic interface for all FCMs and consists of approximately 35 methods. Each FCM class from which instances can be created is used to define the page fault and management behaviour for specific memory mappings (i.e. `FCMDefault` for standard mapped files, `FCMComputation` for anonymous mappings and others for specialized mappings, such `FCMLTransTable` for the local clustered object tables.) The primary branch of the tree from which the majority of runtime instantiations occur are the subclasses of FCMCommon. The classes utilize implementation inheritance in order to ease development and maintenance. Doing so, however, leads to a locking discipline that cuts across all the classes of the hierarchy. This introduces subtle inter-dependencies which makes it very difficult to add a new leaf class that inherits functionality but isolates a given path for reimplementation with different synchronization semantics.

Given the relationships among the existing classes and complexity inherent to the FCM implementations, a new distributed FCM could not be easily derived from the shared implementations. However, it did not seem prudent to invest in the construction of a new distributed FCM without gaining some initial experience and insight to justify the effort. As such, an initial attempt was made to validate the use of distribution in the FCM on the page fault path by constructing a simplified, crippled `FCMPartitionedTrivial` class which only implemented enough of the FCM

interface to allow a hand-crafted test case. The test permitted the performance of multiple faulting threads on a dummy region to be examined. Alleviated from the burden of complete functionality, only four methods required implementation. However the resulting FCM bears little resemblance to standard FCMs either in implementation or functionality, as it:

- requires custom hand instantiation and mapping,
- only provides zero filled pinned memory,
- does not support IO interaction with filesystem (ie. does not implement asynchronous IO required for swapping functionality)
- does not support blocking faults,
- cannot be destroyed,
- does not support page reclamation, and
- does not support fork semantics.

For hand constructed experiments, in which multiple threads concurrently accessed a region of memory, backed by an instance of `FCMPartitionedTrivial`, improved scalability was observed, in comparison to `FCMDefault`. Having validated with simple techniques the benefit of using distribution on the page fault path, we were motivated to utilize distribution in the standard FCMs. It is not clear, for a complex object which is on multiple system paths that it is possible to optimize one path using distribution without degrading other paths, such that end system performance is improved. Our options were to either completely redesign and implement the FCM hierarchy with distribution in mind or attempt to somehow integrate distribution in the current scheme. The former approach provides much greater latitude in dealing with multiple conflicting paths as distribution can be more fundamentally accounted for via algorithmic redesign. However, given the nature of complex core system services, redesign has a number of drawbacks:

- Introducing distribution to a complex service usually only adds more complexity.
- Confidence and familiarity in "working software" can be hard won and should not be sacrificed lightly. Maintenance and debugging of fundamental system services is a key challenge.

Given the above drawbacks, it was decided that an incremental approach would be preferable if it preserves the fundamental protocols of the current design while enabling higher performance. If

this approach failed to yield an implementation which was capable of delivering end performance benefits, the knowledge gained would better guide a complete redesign.

After several failed attempts to directly integrate distribution in the current implementations, a more modular approach was adopted. A single distributed hash table component was developed which encapsulated the distribution of the table and associated locking semantics with a well defined interface. The hash table is not a Clustered Object itself, but rather is designed to be embedded within a Clustered Object. We refer to such components as Clustered Object Constituents or simply constituents (see Section 5.2). The distributed hash table constituent is referred to as *Dhash*. By encapsulating the locking and distribution with a well-defined interface, it was possible to better isolate the changes and enable re-implementation of the current FCMs, which utilize a hash table as its main data structure. This allowed us to preserve the majority of protocols and structure of the current FCMs. An additional benefit was the development of a general distributed constituent which could be used in the development of other Clustered Objects.

It is worth noting a contradiction often encountered when implementing system level code. On the one hand, the demand for performance leads one to explore more complex implementations, such as a distributed FCM, however to ease the burden, a generalized component, such as Dhash, is desirable even though such components often sacrifice performance due to their general nature. OS developers often avoid using general libraries when optimizing, as the need to exploit features of the specific problem at hand precludes the generalized software. However, we contend that encapsulated, highly concurrent, distributed implementations, which can ease optimization efforts, will permit OS developers to explore aggressive multi-processor optimization in an incremental fashion, thus enabling optimizations that might otherwise be too daunting.

We shall continue our discussion of the optimization of the FCM by first presenting the Dhash constituent that was developed, and its use in the re-implementation of the FCM.

Dhash Overview

Motivated by the goal of optimizing the page lookup operation of the FCM, we developed a distributed hash table which has the following properties:

- provides a set of simple self contained synchronization semantics;
- can be included into a Clustered Object; and
- supports a distributed data model for the data elements of the hash table.

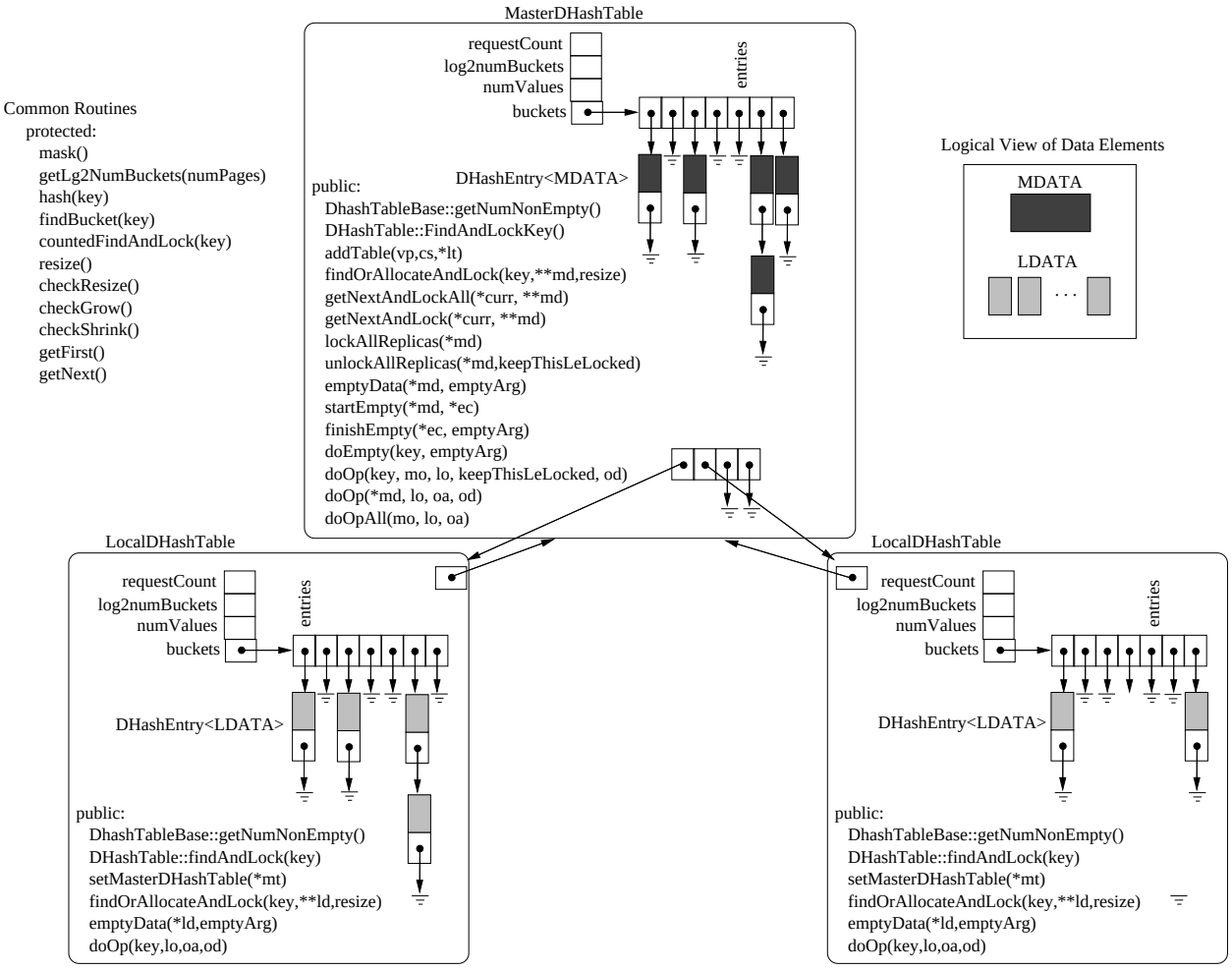


Figure 4.13: An example Dhash structure.

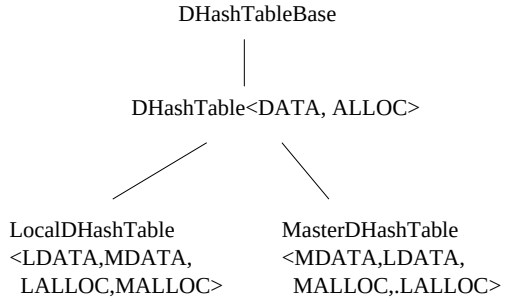


Figure 4.14: Dhash Classes.

Figure 4.13 illustrates the basic structure of an instantiated Dhash constituent and Figure 4.14 the associated class hierarchy. There are two basic components to the Dhash: a MasterDhashTable and LocalDhashTables, which are designed to be embedded into a Clustered Object's root and Representatives, respectively. After embedding the Dhash into a Clustered Object, calls can be made to either the LocalDhashTables or the MasterDhashTable directly. Dhash has a number of interesting features:

1. LocalDhashTables and MasterDhashTable automatically cooperate to provide the semantics of a single shared hash table for common operations, hiding its internal complexity.
2. All locking and synchronization are handled internally.
3. Fine-grain locking is used, where common accesses require an increment of a reference count and the locking of a single target data element.
4. All tables automatically and independently re-size themselves.
5. Data elements which have both shared and distributed constituents are supported.
6. Scatter and gather operations are supported for distributed data elements.

Each local table has a pointer back to the master table and the master table has pointers to all local tables. The basic model of use is to direct queries to the constituent by invoking the `findOrAllocateAndLock` method of the local tables, with a key. This method looks for the data associated with the key in the local table; if it is found, it locks and returns the item back to the caller. If however the item is not found, it searches the master table. If the data element is found in the master table, it is locked and a local data copy is made, placed into the local table, locked and returned to the caller, with the master data element being unlocked prior to return. However, if the data element is not found in the master table, a new master data entry is allocated, prior to the local copy being made. There are obviously many potential race conditions and complexities in the above description (remembering that the allocations may also cause resize of the tables independently). The code for the local and master tables has been carefully constructed to deal with these conditions. It is important to note that the distributed nature of the lookup is hidden from the caller. The caller is always guaranteed to be returned a locked local data element and an indication of whether the element was newly allocated or found.

The Dhash constituent supports a distributed data model where data elements can have global data and operations, as well as local data and operations. This allows the implementor to utilize

Dhash to implement optimized distributed data models. The data being stored in the hash table might have operations which can operate on local data elements that do not require coherence with the other copies. However, other operations might require that certain state be global to all copies. For example, we could design a page descriptor which tracks the state of a physical page frame in a distributed fashion, where locally the descriptor tracks whether the page has been mapped to an address space for access. Doing so allows concurrent faults on the page to proceed without having to lock a shared descriptor. However, this implies that, to determine if a page has been mapped into an address space, aggregation across all of the local state of the page descriptor is required. For page descriptors, this is a relatively infrequent operation and allowing faults to occur in parallel is a greater win. Conversely, the physical address of the page frame is clearly a global fact to the page descriptor and thus one would like to store it globally and ensure that all local copies of the descriptor access the global value.

The MasterDhashTable provides master operations which allow both global and distributed operations on the data elements to be implemented. General support for distributed data allows the constituent to be used in a diverse set of scenarios. It can be used in a more traditional fashion in which both the global and local tables simply associate a key with a pointer to a shared data record, or it can be used to implement more complex distributed data elements in which a key is associated with replicated data which has specialized data coherence and distributed semantics.

Dhash Interface and Semantics

The operations of Dhash can be broken down into three categories:

Optimized lookup: This is the primary interface for the lookup service provided by Dhash. It provides an efficient and aggressive distributed lookup service with which a client can query and store data into the table associated with a key:

`LocalDhashTable::findOrAllocateAndLock` and `MasterDhashTable::findOrAllocateAndLock`: When the local version is invoked, the caller is ensured to have a local data element, associated with the specified key, returned in a locked state. A data element is considered to be a holder of the target data. If there is no data present in the table for the key, the data element returned is in the empty state but locked to provide race free insertion of data. The return code indicates that either a non-empty or empty data element was returned. Only one caller will ever get back a return status indicating empty and

1. increment request count on local table, this ensures that the local table will remain stable.
2. lookup key in local table
3. if found:
   (a) lock element
   (b) re-confirm match as element was not locked when initial match was made; on failure unlock element and restart search from 2.
   (c) decrement request count of local table
   (d) return element and set return status to found
4. else if not found locally
   (a) decrement request count of local table
   (b) invoke findOrAllocateAndLock of master table; regardless of whether the data was present in the master table or not a locked master data element associated with the key is returned. However the return status does indicate if our query caused the allocation of the element or whether it already existed in the master. The head of the hash bucket is used to synchronize insertion using atomic primitives thus ensuring that only one thread ever allocates a new element.
   (c) increment local request count
   (d) search local table again to see if the element has been added while we were searching the master
   (e) if it was not found
      i. allocate a local data element and lock it
      ii. decrement local request count
      iii. replicate the master data element to the local element
   (f) else lock the element and re-confirm match; on success decrement local count; on failure unlock element and restart from 4.(d)
   (g) unlock the master element
   (h) return the local data element and set the return status to indicate if the element was found or allocated in the master

Figure 4.15: Basic algorithm for LocalDhashTable::findOrAllocateAndLock (does not include table resizing)

can thus be ensured that it can atomically take the necessary actions to fill the data element. Internally, Dhash synchronizes around the associated master data element for the key. The algorithm is outlined in Figure 4.15. Despite the subtle synchronization nature of the algorithm, the two properties worth noting are: 1) in the case of the element being present, all that is required is an increment of the local request count, hash lookup, lock of the element and a final decrement of the local request count; and 2) the subtle internal synchronization is not exposed to the caller, the caller, regardless of what happens, is ensured to have a local data element returned and that the return code will indicate if they are responsible for filling the data element. The master version of `findOrAllocateAndLock` is similar, but deals with the master table and master data elements. It can be directly invoked if a client wants to query the master table, or more commonly invoked by the local version, as indicated in Figure 4.15.

Data element management: These methods are provided for clients to manage the distributed nature of the data elements in a systematic way while isolating the complexity of the synchronization:

`MasterDhashTable::doOp`: This is the primary function for applying an operator to a data element in a distributed fashion. Given a master and local operator as well as locking directives, this method applies the operators to the specified data element while encapsulating all necessary synchronization. Specifically they can be used to implement scatter and gather functions.

`MasterDhashTable::lockAllReplicas` and `unlockAllReplicas`: Given a locked master data element these methods can by used to explicitly lock or unlock all its replica's. This allows the construction of loops which atomically manipulate the replicas, outside of the doOp support.

`MasterDhashTable::emptyData`: This method allows a client to remove a data element by resetting the data element to the empty state. Internally, the method ensures that the empty occurs atomically across all replicas of the element. It also ensures that the data element is destroyed or reused only when it is actually safe to do so, that is, no threads are actively inspecting the associated memory.

`MasterDhashTable::startEmpty`, `finishEmpty` and `doEmpty`: In some scenarios clients may require more control over the removal of a data element from Dhash. These methods

provide this control while still ensuring that the basic internal synchronization rules are obeyed.

Table management: These methods are provided for clients to iterate over all elements the hash tables.

`MasterDhashTable::doOPAll`: Similar to doOp above but applies the specified operators across all data elements present in the tables.

`MasterDhashTable::addReference` and `removeReference`: These are utilities which allow a client to construct iterations over all the data elements. A client must bracket their function with addReference at the beginning and removeReference at the end. The following methods can be used to implement the actual iteration over the elements.

`MasterDhashTable::getNextAndLockAll`: Once a client has invoked addReference, this method can be used to iterate over the data elements. This method ensures that all the replicas of the data elements are locked upon return.

`MasterDhashTable::getNextAndLock`: Similar to above but only locks the master and not all the replicas of a data element.

Improvements

One of the advantages of the constituent approach is that we can progressively make improvements without having to modify the Clustered Objects that use Dhash. There are three main improvements to the Dhash table implementation which have been identified to date:

1. A static fixed size array is used in the master to maintain pointers to the local tables; a generalization would be to utilize a dynamic array.

2. Currently request counters are used to guard the hash table data structures and ensure that data elements are not prematurely destroyed. The read-copy-update facilities (see 5.4.3) of K42 should be used to guard the hash table data structures eliminating the use of request counters.

3. The data element empty methods require further optimization; empty elements should be more aggressively removed from hash chains and be made eligible for reuse.

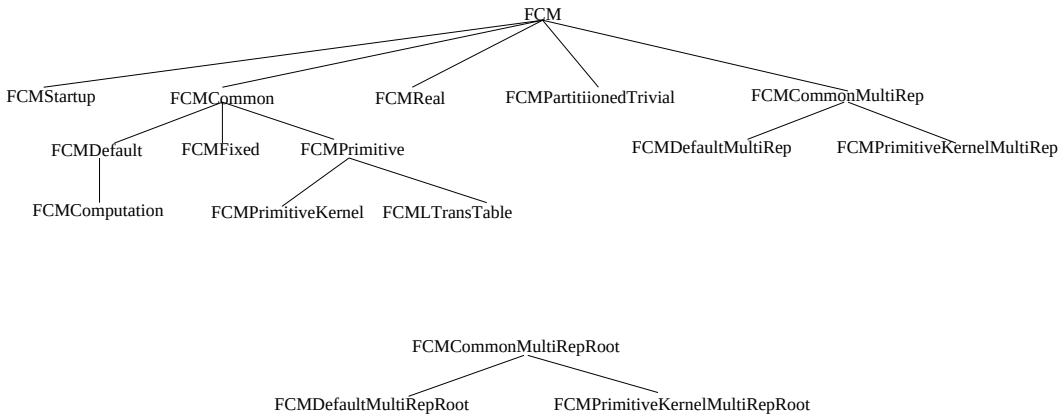


Figure 4.16: New FCM Class Hierarchies

The current Dhash implementation, without the above improvements, is sufficient to re-implement the FCM. The improvements further generalize the utility of Dhash and potentially optimize the base lookup performance, with respect to cycles. We believe all changes can be made as future work, without impacting client code given the encapsulation afforded by the modularity.

Dhash FCM

As the system has evolved, various performance requirements on the FCM have been recognized. Prior to support for shared segments (the use of common page tables when sharing mapped files such as executables), there was considerable concurrent demand on the FCMs for shared files even in single threaded multi-user workloads[6]. Consider the text of a common executable such as a Unix shell. A process created to run the shell suffers page faults on the processor it is executing on to establish the page frames of the executable in its page tables. As such, the FCM backing the shell's text pages will suffer page faults on that processor, even though the pages will have likely already been accessed on the same processor due to a prior run. This original scenario motivated us to develop a distributed FCM based on Dhash, in which the hot path was the remapping of the resident pages of a long lived file. An FCM implementation using the Dhash constituent in a straightforward manner is sufficient in this case, as the majority of page faults will result in accesses only to the local Dhash table in order to remap pages which have already been mapped on the processor by a previous execution. This scenario can be highlighted by a micro-benchmark, in which an instance of grep is run on each processor searching a common file. The instance of

[6]Although there is support for shared segments, which can alleviate remapping requests to the FCMs which back them, there are still scenarios which prohibit us from using shared segments under various conditions. For example, the text of a Linux executable compiled on non-K42 systems cannot use shared segments.

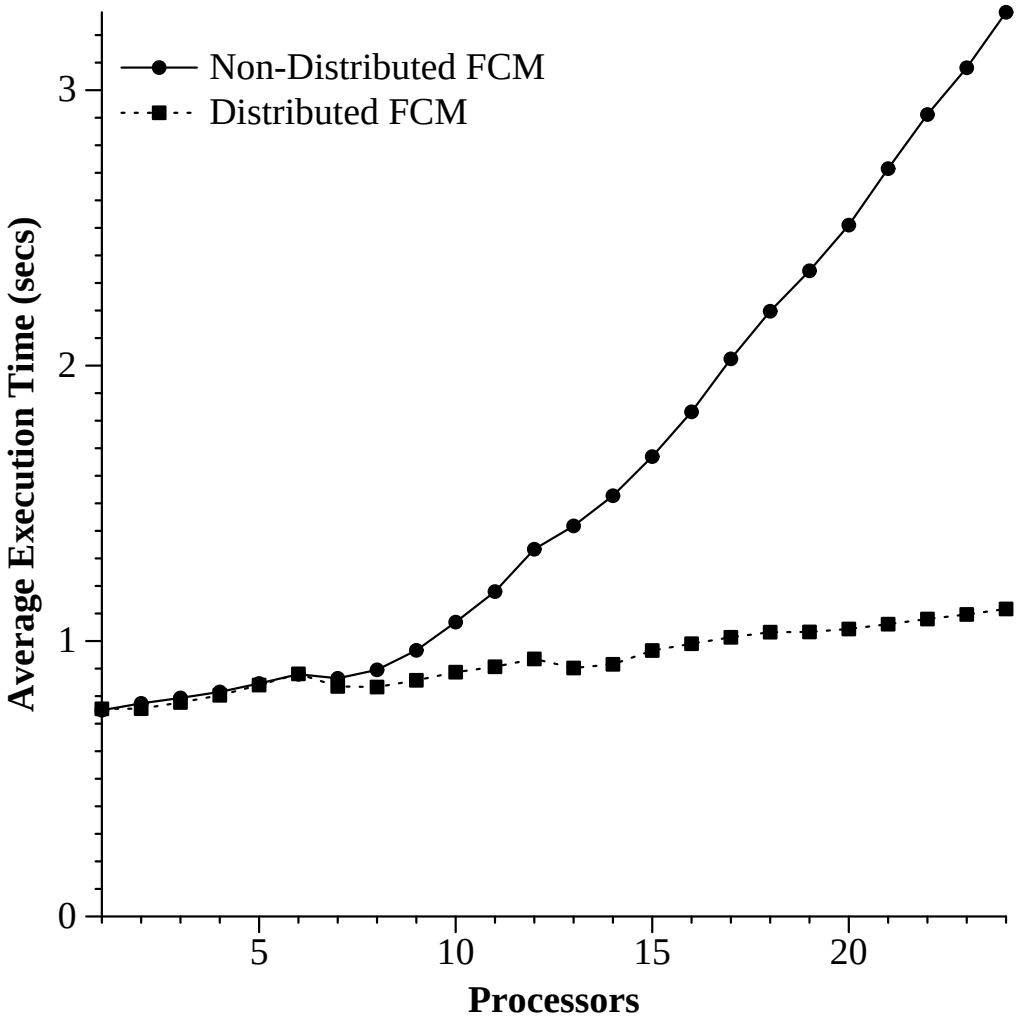


Figure 4.17: Graph of Distributed FCM vs Non-Distributed FCM performance: this graph illustrates the results of running one instance of grep per processor. Each grep searched a common 111MB file for a common non-matching search pattern. We measured the average time for an instance of grep to complete.

grep will induce concurrent page faults to the FCM, which caches the pages of the executable as well as concurrent page faults to the data file which is being searched. Figure 4.17 illustrates the performance of such a micro-benchmark using the non-distributed and the distributed version of the FCM.

When we consider the performance of the modified version of memclone (Figure 4.18) with the distributed FCM (line labeled Distributed Dhash FCM), we observe poor scalability. The distributed FCM was designed to optimize the resident page fault path, in which the majority of faults would be to pages which already exist in the FCM and have been previously accessed on the processor. The memclone benchmark does not have this behaviour. Most of its page faults are to pages which do not exist in the FCM, but do not require IO since they are initial touches

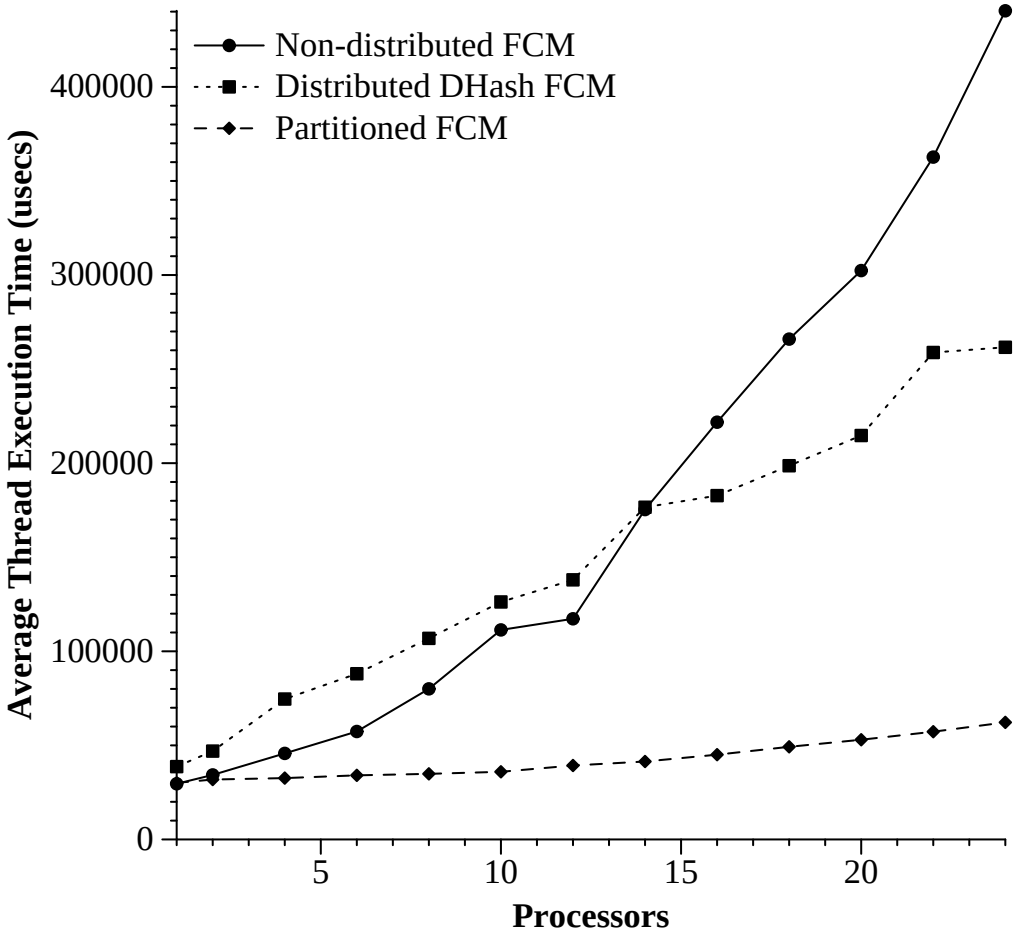


Figure 4.18: Graph of average thread execution time for a modified memclone running on K42. One thread is executed per-processor. Each thread touches 2500 independent pages of a single large common memory allocation. Each page touch results in a zero-fill page fault in the OS. The graph plots the performance of using a non-distributed FCM versus a distributed FCM using Dhash versus a partitioned FCM, to back the single common memory allocation. Ideal scalability would be represented by a horizontal line.

of anonymous (zero filled) memory. This results in the page faults of memclone missing in the local tables of Dhash and accessing the Dhash master table. The distribution of Dhash has no real benefit, therefore, since all initial faults will go through the central master table, and we expect no additional faults to an individual page. The distributed version acts like a central fine grain locked hash table with the additional overhead of creating replicas whose cost of creation is never amortized over multiple accesses. These costs are only justified if there are additional faults to the pages which are satisfied from the replica created in the local hash tables. As noted above, this can occur when a file is remapped on a processor, as in the case of an executable. It could also occur because the Representatives of the FCM are assigned to more than one processor and thus the local hash tables service a set of processors. In this case, access by one processor will enable cheaper

access for the other processors sharing the Representative. It is left as future work to explore this scenario for randomly accessed heaps of a long-lived system server (or server application such as a database).

We can see from these examples that an individual distributed implementation optimized for a given scenario may not be ideal for all usage scenarios. This is perhaps intuitive, as a distributed implementation typically identifies a single path for optimization in order to permit more aggressive techniques. Instances of a core system component, such as the FCM, can be used in multiple scenarios in which the path for which it was optimized is not the hot path; however, it has been our experience so far that a single instance is not typically used in multiple scenarios simultaneously. For a specialized application like that modeled by the modified memclone benchmark, which has unique parallel requirements for a single region of its address space, a separate distributed implementation can be used. It is not clear, however, if there is a general solution for automatically identifying when to use one implementation over another.

In the case of parallel scientific applications, such as those modeled by memclone, it is not unreasonable to provide an interface which permits the application to specify that a given memory allocation should be optimized for parallel access and thus yield better performance. The general issues of how to identify when one implementation should be used rather than another is beyond the scope of this thesis – rather our main concern is to illustrate that distributed implementations are feasible. In Chapter 6 we present a facility for the dynamic switching of a Clustered Object implementation which could be used to permit the selection of one implementation over another at arbitrary points in the execution of a workload rather than just at instantiation time.

In the modified version of the memclone micro-benchmark, the data array is backed by a single computational region whose individual pages are accessed predominately on a single processor and only suffers an initial fault. A distributed FCM which utilizes a partitioned hash table would be sufficient to improve the performance. A partitioned FCM could utilize parts of the Dhash constituent, but in a different fashion. This version would essentially partition the set of pages across a set of Representatives, but not distribute the page descriptors. To do this, a fixed number of m master Dhash tables are used to manage the page descriptors for a fixed disjoint set of pages. A new local Dhash table class would be constructed which simply redirects invocations to one of the m master hash tables. All Representatives would have an instance of the local Dhash table but only the first m Representatives have a master hash table. The policy for page assignment can be determined by a specified shift value which is applied to the page offset. The result is then mapped

to one of the m master tables via modulo calculation. Such an implementation would require the degree of parallelism and, optionally the size of the region, to be known when it is constructed. A prototype of the partitioned FCM was constructed and its performance is also shown in Figure 4.18.

Other uses of Dhash

We have identified two other system services that are candidates for re-implementation using the Dhash constituent. A full description, evaluation, and re-implementation of these objects is beyond the scope of this thesis. A brief description of the objects is provided below.

User Level PThread Support Each address space in K42 implements its own K42 thread level scheduler on top of which a standard Posix thread service is implemented. As part of this implementation, a service to map a Posix thread identifier to an underlying K42 thread descriptor is required. Currently a simple centralized locked global array is used to achieve this mapping. We believe that this service can be reimplemented with the Dhash constituent in order to alleviate lock contention and promote locality on Posix thread services such as creation and destruction paths which require manipulation of the array. Furthermore, Posix thread delivery services require lookup into the array although this does not require manipulation of the array. Currently the global lock is used to synchronize these operations as well, which would also benefit from a Dhash based implementation. Although other solutions could be used, given the encapsulated nature of the Dhash constituent, we believe that its aggressive fine grain locking nature can be utilized with less effort.

File System Pathname Translation To alleviate the burden of filesystem development in K42, a set of common file system services, implementing path to file object translation and locking semantics, is provided. Currently these services are implemented as a tree of directory based hash tables which are protected by locks associated with each hash table. We believe that a new single level direct mapping service can be implemented based on the Dhash constituent. Dhash's flexible locking semantics and highly concurrent nature would permit us to use a single hash table which translates a complete path string to a file system object handle.

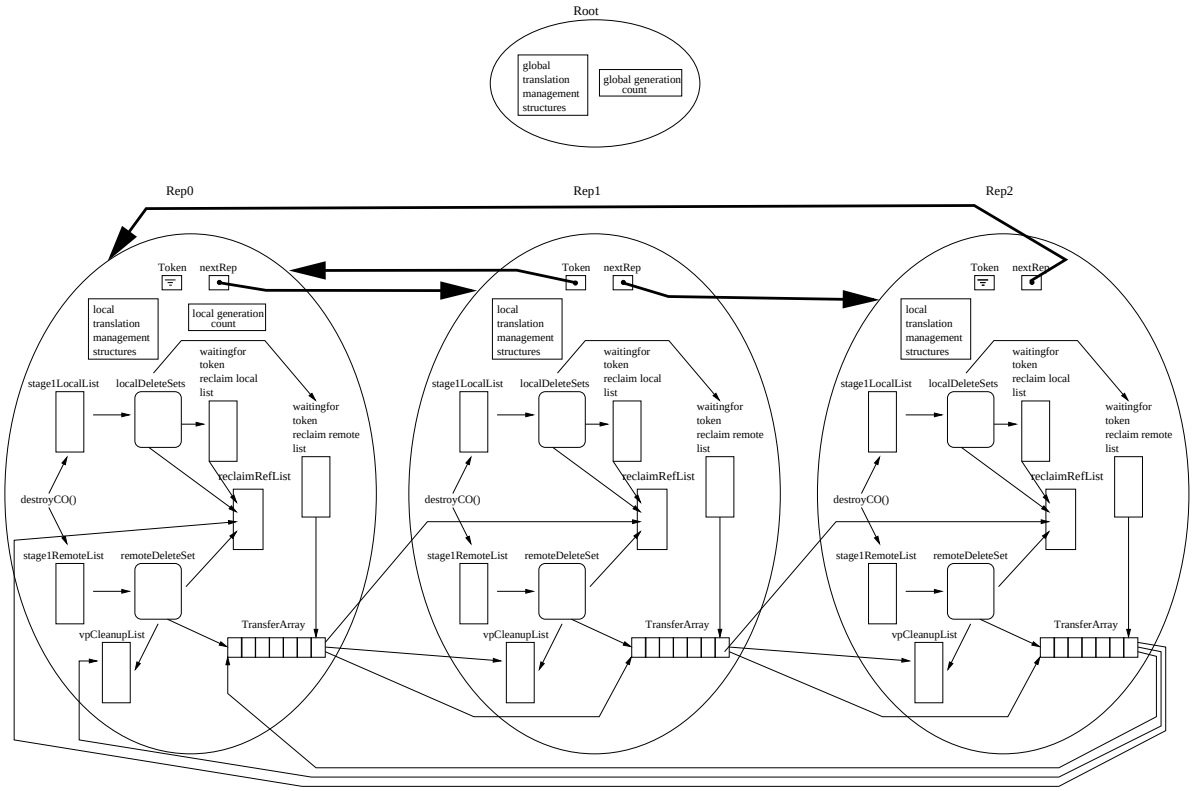


Figure 4.19: Clustered Object Manager

4.2 Clustered Object Manager

In K42, we expect that some services will be developed from their inception with distributed implementations. An example of this is the Clustered Object infrastructure which was designed to be scalable using distributed algorithms and an implementation which avoids global intra-processor communication. One of the key components is a Clustered Object called the Clustered Object System Manager (COSMgr) which we developed from its inception as a distributed Clustered Object with multiple Representatives. The COSMgr of each address space provides Clustered Object allocation and destruction services, and, unlike the objects previously described, was not incrementally distributed.

Figure 4.19 illustrates the basic structure of the COSMgr. Rather than discussing the details of the implementation, at this point we simply note the salient features with respect to distribution. In the next chapter we discuss some of the services it provides and algorithms it employs in more detail.

Each Representative is designed to serve two roles:

1. Provide local services of the Clustered Object infrastructure to requests from the processor with which it is associated. Such requests are satisfied in a fashion that requires no coordination or communication with other Representatives. Examples of these include the allocation of a Clustered Object identifier and the reclamation of memory associated with a Clustered Object which has been allocated and only accessed on the local processor.

2. Participate in the distributed services of the COSMgr by communicating and synchronizing between Representatives in a pair-wise manner. Features of these distributed operations include:

   - No operations require global iteration over the Representatives, rather the Representatives are linked together in a circular structure (dark arrows in figure).
   - Data flows (represented by thin arrows in Figure 4.19) are predominately internal to a Representative, and those that are not internal are only between neighboring Representatives.
   - Token passing is used as the primary mechanism for distributed synchronization, and is implemented using cache lines which are at most shared between two processors.[7]
   - Along with the token, per-Representative transfer arrays are used to pass distributed work among the Representatives. Work which requires distributed computation is placed onto a Representative's transfer array. A Representative's transfer array is read by the next Representative in the circular structure when the token is passed to it. The Representative that receives the token reads the work request from the transfer array of the previous Representative which had the token and processes the requests. Requests which still require further distributed processing are similarly placed on its transfer array to be passed to the next Representative when the token is passed. In this manner, distributed computations are accomplished by passing work requests between Representatives in a pair-wise fashion.

Another Clustered Object that has been developed from, scratch with multiple Representatives, (by a member of the IBM K42 team) is a Resource Manager which tracks local resource usage in order to implement distributed algorithms for management, including load balancing and admission control.

[7]The token passing mechanism is invoked periodically at a low frequency. Alternate feedback driven designs which invoke token circulation only when necessary and on processor subsets have been considered but not explored.

In 2004 two new distributed Clustered Objects which factor out key aspects of the current COSMgr, an Event Manager and a Read-Copy-Update (RCU) Manager were being developed. These Clustered Objects optimize and generalize aspects of the current garbage collection mechanisms in order to permit more general use. The Event Manager is intended to implement a distributed communication service, using distributed event queues and local soft timer [6] techniques, in order to provide a light-weight batching inter-processor communication facility. The RCU manager is intended to provide the current thread tracking mechanisms of the COSMgr in a more general, non-polling, fashion in order to facilitate wider use of these mechanisms. The development of and evaluation of these Clustered Objects is outside of the scope of this thesis. RCU techniques are briefly discussed in the next chapter.

4.3 Summary

In this chapter we have presented examples of using distribution in a modular fashion in K42. We have focused on improving locality of objects in the virtual memory management of K42, to yield scalable resident page fault performance. In the next chapter we will discuss the Clustered Object model and the infrastructure used to construct these examples. Three key points from this chapter are:

- The complexity of a distributed implementation can successfully be encapsulated by a Clustered Object.
- Multiple objects may require distributed re-implementations in order to improve the scalability of a given service. The Clustered Object approach successfully supports incremental evaluation and development.
- Simple caching strategies can be used to ease the burden of developing distributed implementations of system objects, however, a given distributed implementation can be considerably more complex than non-distributed versions.

Chapter 5

Clustered Objects

In this chapter we describe the details of the Clustered Object protocols and infrastructure we have developed. We begin with a overview of how Clustered Objects are implemented.

5.1 Overview

Each Clustered Object is identified by a unique interface to which every implementation conforms. We use a C++ pure virtual base class to express a Clustered Object interface (Clustered Object interface class). An implementation of a Clustered Object consists of two portions: a Representative definition and a Root definition, expressed as separate C++ classes. The Representative definition of a Clustered Object defines the per-processor portion of the Clustered Object. In the case of the performance counter, it would be the definition of the sub-counters. An instance of a Clustered Object Representative class is called a Rep of the Clustered Object instance. The Representative class implements the interface of the Clustered Object, inheriting from the Clustered Object's interface class. The Root class defines the global portions of an instance of the Clustered Object. Every instance of a Clustered Object has exactly one instance of its Root class that serves as the internal central anchor or "root" of the instance. Each Rep has a pointer to the Root of the Clustered Object instance. The methods of a Rep can access the shared data and methods of the Clustered Object via its root pointer.

At run-time, an instance of a given Clustered Object is created by instantiating an instance of the desired Root class.[1] Instantiating the Root establishes a unique *Clustered Object Identifier*

[1]The client code is not actually aware of this fact. Rather, a static *Create* method of the Rep class is used to allocate the root. Because we do not have direct language support, this is a programmer enforced protocol.

(COID also referred to as a Clustered Object ref) that is used by clients to access the newly created instance. To the client code, a COID appears to be a pointer to an instance of the Rep Class. To provide better code isolation, this fact is hidden from the client code with the macro: `#define DREF(coid) (*(coid))`. For example, if `c` is a variable holding the COID of an instance of a clustered performance counter that has a method `inc`, a call would look like: `DREF(c)->inc()`.

A set of tables and protocols are used to translate calls on a COID in order to achieve the unique run-time features of Clustered Objects. There is a single shared table of Root pointers called the Global Translation Table and a set of Rep pointer tables called Local Translation Tables, one per processor. The virtual memory map for each processor is set up so that a Local Translation Table appears at address `vbase` on each processor but is backed by different physical pages[2]. This allows the entries of the Local Translation Tables, which are at the same virtual address on each processor, to have different values on each processor. Hence, the entries of the Local Translation Tables are per-processor despite only occupying a single range of fixed addresses. When a Clustered Object is allocated, its root is instantiated and installed into a free entry in the Global Translation Table. The Global Translation Table entries are managed on a per-processor basis by splitting the global table into per-processor regions of which each processor maintains a free list and only allocates from its range, avoiding synchronization or sharing. The address of the corresponding location in the Local Translation Tables address range is the COID for the new Clustered Object. The sizes of the global and local tables are kept the same, allowing simple pointer arithmetic to convert either a local to a global or a global to a local table pointer. Figure 5.1 illustrates a Clustered Object instance and the translation tables.

In order to achieve the lazy creation of the Reps of a Clustered Object, Reps are not created or installed into the Local Translation Tables when the Clustered Object is instantiated. Instead, empty entries of the Local Translation Table are initialized to refer to a special hand-crafted object called the Default Object. Hence, the first time a Clustered Object is accessed on a processor (or an attempt is make to access a non-existent Clustered Object), the same global Default Object is invoked. The Default Object leverages the fact that every call to a Clustered Object goes through a virtual function table. (Remember that a virtual base class is used to define the interface for a Clustered Object.) The Default Object overloads the method pointers in its virtual function table to point at a single trampoline[3] method. The trampoline code saves the current register state

[2]In K42, a page table is maintained on a per-processor and per-address space basis, and thus each processor can have its own view of the address space.

[3]Trampoline, refers to the redirection of a call from the intended target to an alternate target.

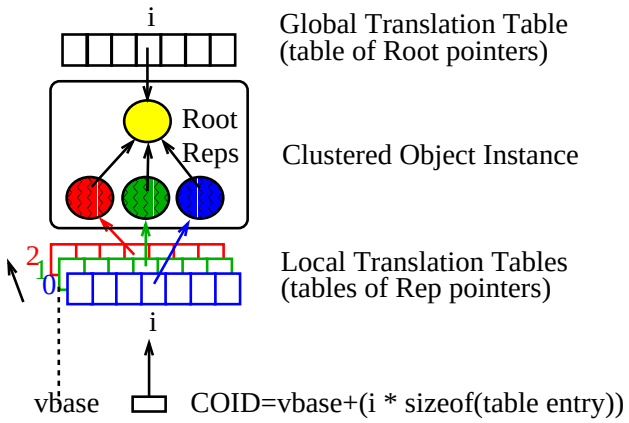


Figure 5.1: A Clustered Object Instance and Translation Tables.

on the stack, looks up the Root installed in the Global Translation Table entry corresponding to the COID that was accessed, and invokes a well-known method that all Roots must implement called `handleMiss`. This method is responsible for installing a Rep on the processor into the Local Translation Table entry corresponding to the COID that was accessed. This is done either by instantiating a new Rep or by identifying a preexisting Rep and storing its address into the address pointed to by the COID in the Local Translation Table. On return from the `handleMiss` method, the trampoline code restarts the call on the correct method of the newly installed Rep. The above process is called a Miss and its resolution Miss-Handling. Note that after the first Miss on a Clustered Object instance, on a given processor, all subsequent calls on that processor will proceed as standard C++ method invocations via two pointer dereferences. Thus, in the common case, methods of the installed Rep will be called directly with no involvement of the Default Object.

The map of processors to Reps is controlled by the Root Object. A shared implementation can be achieved with a Root that maintains one Rep and uses it for every processor that accesses the Clustered Object instance. Distributed implementations can be realized with a Root that allocates a new Rep for some number (or cluster) of processors, and complete distribution is achieved by a Root that allocates a new Rep for every accessing processor. There are standard K42 Root classes which handle these scenarios.

5.2 Clustered Object Development Strategies

Rather than having each Clustered Object developed in an ad hoc way, we expect that a library of reusable distributed data structures, which provides standard abstractions, will be developed over

time to facilitate the wider development of distributed Clustered Objects. Such data structures should provide a well defined and flexible set of base components for Clustered Object development. They should encapsulate the complexity of the distributed protocols associated with the data structure, governing the interaction between the Representatives, and between a Representative and the Root. These base Clustered Object constituents should allow the programmer to optimize the behaviour and usage for the specific Clustered Object being built.

In general we expect there to be two main paths to Clustered Object development:

1. Incremental approach:
   (a) Start with a more traditional shared implementation.
   (b) Identify hot contended paths and operations (given a set of operating parameters).
   (c) While preserving the majority of the shared implementation, introduce distribution to alleviate sharing on the hot operations. The fundamental algorithms and paths in which the object participates remain the same.
   (d) Run like this until new operating parameters dictate that a "from scratch distributed design" is necessary.
2. From scratch approach:
   (a) Designing the paths and algorithms in a distributed fashion utilizing a distributed implementation.

Along both paths, we expect Clustered Object constituents to alleviate some of the development burden, particularly on the incremental path. However, it is expected that some objects will require more extreme performance optimizations requiring completely hand written implementations.

5.3 K42 Clustered Object Facilities

As part of this research effort, we have implemented basic Clustered Object infrastructure within K42, as motivated by our prior work in Tornado [49], which includes the translation tables and associated mechanism which enables the miss-handling process as well as the Clustered Object garbage collection mechanisms (not yet described). Unique to this thesis, we have developed a model for Clustered Object development and a set of protocols to generalize the use of the basic

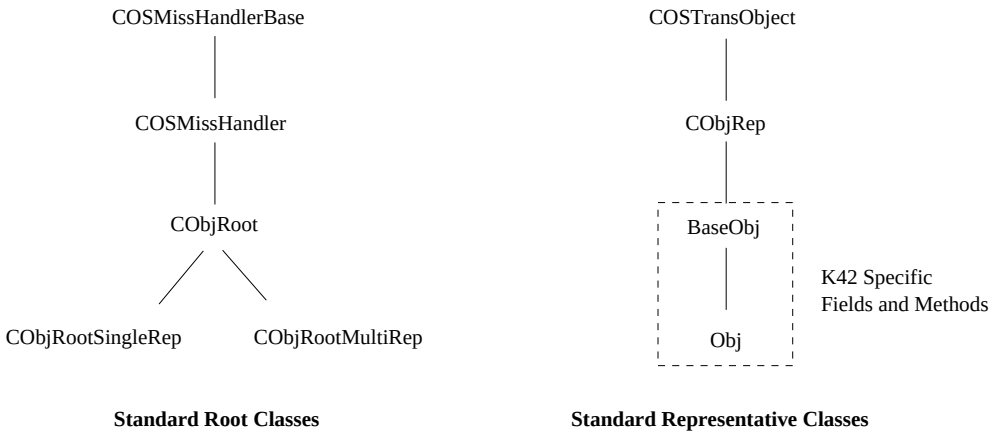


Figure 5.2: Standard Clustered Object Base Classes for Roots and Representatives.

facilities. This includes higher level distributed garbage collection facilities[4] as well as a set of standard base C++ classes for the development of Clustered Objects. Apart from the standard base classes, we will refer to the Clustered Object infrastructure and services of K42 that have been developed, as the Clustered Object System (COS). The COS and base classes are designed to cooperatively define and provide the Clustered Object model and protocols. The majority of new COS function is implemented by a distributed Clustered Object called the Clustered Object System Manager (COSMgr), briefly described in Section 4.2.

5.3.1 Clustered Object Root Classes

Figure 5.2 illustrates the class trees that were developed.

On the left side of Figure 5.2 is the class tree that enables the development of Clustered Object roots. The top two classes (`COSMissHandlerBase` and `COSMissHandler`) define the common interfaces that all Clustered Object roots require to properly interact with the basic miss-handling mechanisms and garbage collection protocols of the COS. Specifically, they define the interfaces that the COS expects to invoke during miss-handling and deallocation (garbage collection):

handleMiss: As described in Section 5.1, the `handleMiss` method is invoked by the translation mechanisms, during the miss-handling procedure, on the root of the Clustered Object on which the miss occurred.

[4]Gamsa proposed and implemented the basic scheme for shared object destruction in Tornado including a token based algorithm for the establishment of system wide quiescent states. In this work, I have implemented a redesigned scheme focusing on the destruction of distributed Clustered Object implementations. The new mechanisms and protocols are as described in this dissertation.

getRef: Every root must be able to return the Clustered Object identifier associated with the Clustered Object it is part of. This service is provided by the `getRef` method.

getTransSet: During deallocation of a Clustered Object, the COS invokes the `getTransSet` method to obtain the translation set from the root of the Clustered Object. Every Clustered Object is responsible for tracking the set of processors for which it has established a translation.

getVPCleanupSet: In order to ease reclamation of the distributed resources of a Clustered Object, the COS provides a facility, which a developer can use to request asynchronous call-backs of the Clustered Object's `cleanup` method (described next). To specify that call-backs should occur and the set of processors on which they should occur on, the developer implements the `getVPCleanupSet` method. This method is invoked by the COS when the Clustered Object is submitted for deallocation, which ensures that call-backs will then be invoked on the set of processors returned. If an empty set is returned then only one invocation of `cleanup` method will occur and the Clustered Object must implement its own procedure for distributing the freeing of its memory.

cleanup: This method is invoked by the COS after it has determined there are no threads which can access the memory of a Clustered Object after it has been submitted for deallocation. The object should use this call to free its memory resources. Using the `getVPCleanupSet` method described above, a Clustered Object can have the COS invoke the cleanup method in parallel on a set of processors, in order to implement distributed freeing of its resources.

In addition, the `COSMissHandler` class defines a field called `myRef` for a root to store the Clustered Object identifier for the Clustered Object instance it represents. This is the value that the default implementation of the `getRef` method returns.

The `CObjRoot` class defines common interfaces and default implementations of methods that the standard root implementations (its subclasses) rely on. Specifically, it defines constructors which ensure that, when a root is instantiated, the appropriate methods of the COSMgr are invoked to allocate a new Clustered Object identifier and to have the new root installed as the root for the identifier. These implementations store the identifier in the `myRef` member. `CObjRoot` class also defines the interfaces and the default implementations to enable a Clustered Object to operate with the hot-swapping support of K42 (see Chapter 6).

```
class ProcessDistributedRoot : public CObjRootMultiRep {
     PIDType _pid;
public:
     PIDType getPid() { return _pid; }
     void setPID(PIDType pid) { _pid = pid; }
};
```

Figure 5.3: Example code for defining shared data of a Process Object

The `CObjRootSingleRep` and `CObjRootMultiRep` classes are the default implementations for roots of non-distributed (single Representative) and distributed (multi-Representative) Clustered Objects, respectively. One of the key requirements we identified in earlier work was the need for explicitly supporting both shared and distributed data within a Clustered Object [3]. In K42, the root for a distributed Clustered Object is used as the location for shared data, methods, and management of the Clustered Object.

When writing a distributed Clustered Object, the Root for the new Clustered Object is defined as a descendent of *RootMultiRep*. The developer defines any shared data of the Clustered Object as data members of the Root. For example, in the Process Object, it is reasonable to treat the Process Identifier (PID) to be a shared data member, as it is read only and infrequently accessed, and hence would be a member of the Root. (Figure 5.3 illustrates how this would be coded.) This class hierarchy provides a simple development model for defining global data for a Clustered Object instance and yet still allows for the use of inheritance and encapsulation. In order to access the global data members or methods of the Clustered Object, the methods of the Representatives use a predefined macro `COGLOBAL` as in:

```
if (requestingPID == COGLOBAL(getPid())) { ...).
```

It is important to note that the global data members and methods are not externally accessible; only methods of the Representatives can access the methods and data of the root.[5]

The `CObjRootMultiRep` class provides default implementations of all the methods specified by `COSMissHandlerBase` and `COSMissHandler`. It supports the multi-Representative structure as well as additional internal services to the Clustered Objects. Specifically it:

1. handles all low level interaction with the Clustered Object System, implementing the instance side protocols for distributed, miss-handling (lazy instantiation of reps) and garbage collection,

[5]Clients of a Clustered Object only access the object via it's COID and as such can only invoke the external Representative interface. The Root is encapsulated and not visible externally.

2. provides the base support for arbitrary cluster sizes, where a single Representative can be used for a specified number of processors,

3. provides a way for a developer to specify the custom actions that need to be taken to create the unique Representatives for a given Clustered Object, and

4. tracks and provides services for locating and accessing all of the Representatives of a distributed Clustered Object.

These behaviours are implemented by `CObjRootMultiRep` and require no additional programming when developing a new distributed Clustered Object.

We have special-cased the development of non-distributed Clustered Objects by providing the `CObjRootSingleRep` root class. `CObjRootSingleRep` eliminates some of the overheads associated with supporting general multi-Representative Clustered Objects. When developing a non-distributed Clustered Object, the developer does not need to define a new root class but can simply instantiate an instance of `CObjRootSingleRep`.

5.3.2 Clustered Object Representative Classes

The Representatives (reps) of a Clustered Object provide the means for defining the distributed data and methods of the Clustered Objects. Roots, derived from `CObjRootMultiRep`, ensure that a Rep is created for each cluster of processors, with the cluster size being a parameter of the object instance. Thus the data members of the Rep are per-cluster and the public methods service the request from the processors of the cluster it is associated with. On the right-hand side of Figure 5.2 is the tree of base classes for the development of Clustered Object Representatives.

The `COSTransObject` class ensures that all Representative classes have as their first member a C++ virtual function table pointer. This ensures that all Representatives are compatible with translation tables and low-level miss-handling mechanisms, which rely on the first data word of every Representative being a pointer to its virtual function table. This enables the construction of specialized objects, which can serve as a doppelganger for any Clustered Object, intercepting all invocations to a Clustered Object's interface, as in the case of the Default Object (see Section 5.1). The `CObjRep` is the base class for Clustered Object Representatives which inter-operate with both `CObjRootSingleRep` and `CObjRootMultiRep` root classes. `CObjRep` ensures that all classes derived from it contain a pointer to a root which is a sub-type of `CObjRoot`. It also defines a `cleanup` method which the standard roots invoke when they want a Representative to free its resources.

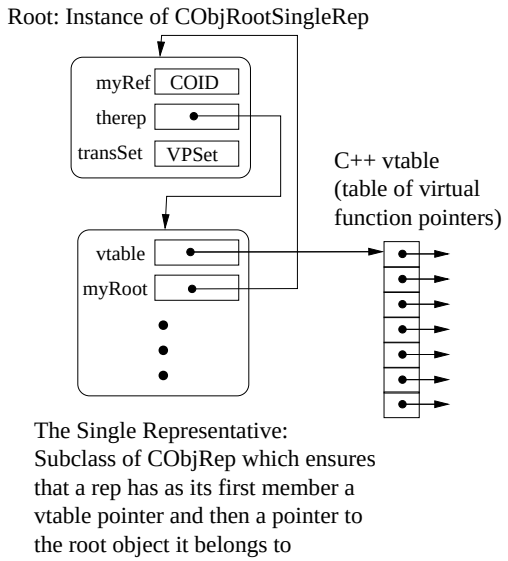


Figure 5.4: A non-distributed Clustered Object instance created from the standard base Clustered Object Classes. It is composed of a single Representative, which is a subclass of CObjRep, and a single instance of CObjRootSingleRep as its root.

The default implementation of this method simply destroys the Representative using the C++ `delete` function, which will trigger the standard C++ destructor for the Representative. We will discuss the destruction protocol in more detail in Section 5.4. The `BaseObj` and `Obj` classes are K42-specific, defining methods and members that a Clustered Object must have to operate with other non-Clustered Object related services of K42.

The `CObjRootSingleRep`, `CObjRootMultiRep` and `CObjRep` classes impose an implicit internal structure for both single-Representative and multi-Representative Clustered Objects. Figure 5.4 depicts a generalized single-Representative Clustered Object, which is composed of a single instance of `CObjRootSingleRep` and a single Representative which is a sub-type of `CObjRep`. In the figure we see that the Root contains three fields:

myRef: As discussed earlier, the root stores the Clustered Object Identifier in this field.

therep: In the case of `CObjRootSingleRep` the root contains a pointer to the single shared Representative, stored in this field.

transSet: The `CObjRootSingleRep` records the set of processors for which it has established translations of the Clustered Object Identifier to the single Representative, stored in this field.

In the figure two fields of single Representative are made explicit:

vtable All Representatives must contain as the first data member the implicit C++ generated virtual function table pointer. We ensure this by carefully constructing the top class (`COSTransObject`) so that all its subclasses will have the vtable pointer as their first member.

myRoot All Representatives contains a pointer to the Root of the Clustered Object instance they belong to.

Figure 5.5 illustrates an example of the code for a simple performance counter implemented as a single-Representative Clustered Object. In K42, we utilize a number of standard macros to control properties of memory used to allocate any C++ object instance. Specifically, for C++ object instances that we know will be accessed only on one processor we use the macro `DEFINE_LOCALSTRICT_NEW`, and for objects instances that will be accessed on many processors we use the macro `DEFINE_GLOBAL_NEW` as in Figure 5.5. These macros make calls to the appropriate K42 memory allocator interface, which minimize false sharing.

Similarly, Figure 5.6 illustrates the structure of a generalized distributed Clustered Object composed of multiple Representatives and a single root, using `CObjRootMultiRep` and `CObjRep` classes. Of particular note is the `replist`, which is used to record the mapping of Representatives to processors. Each node of the `replist` associates a processor (Virtual Processor (vp)) number with a specific Representative. In the next section, we will discuss the implementation of `CObjRootMultiRep` in more detail. Figure 5.7 illustrates an example of the code for a distributed version of the performance counter. It assumes the same base `PerfCounter` class shown at the top of Figure 5.5.

In the distributed version, a root class `PerfCtrDisributedRoot` is defined as a sub-class of `CObjRootMultiRep`. Only the single `createRep` method is defined, which specifies the unique behaviour for the instantiation of Representatives for the distributed performance counter. With respect to the Representative class, the main difference is that the `value` method now is written to implement a gather across the current Rep set using standard methods provided by `CObjRootMultiRep`. The main feature to note is that utilizing the standard base classes, the new classes of the distributed performance counter only had to specify the behaviour unique to the distributed counter and did not have to implement any behaviour specific to the Clustered Object infrastructure.

The key to obtaining good performance is to ensure that the hot or common services of a Clustered Object are serviced by a single rep instance without the need to co-ordinate with the root or other reps, thus only accessing data (including locks) of the Rep to which the access was

```
// Virtual base interface class for Performance Counters
class PerfCounter : public CObjRep {
public:
    virtual SysStatus value(sval &count)=0;
    virtual SysStatus inc()=0;
};
NEW_COID(PerfCounter);

class SharedPerfCounter : public PerfCounter {
protected:
    sval _val;
    DEFINE_GLOBAL_NEW(SharedIntegerCounter);

    SharedIntegerCounter() : _val(0) {}
public:
    static SysStatus Create(PerfCounterCOID &coid) {
        coid=(PerfCounterCOID)
    (new CObjRootSingleRep(new SharedIntegerCounter()))->getRef();
        return 1;
    }

    virtual SysStatus inc() {
        FetchAndAddSignedSynced(&(_value),1);
        return 1;
    }

    virtual SysStatus value(sval &count) {
        count=_val;
        return 1;
    }
};
```

Figure 5.5: Code for a simple non-distributed performance counter.

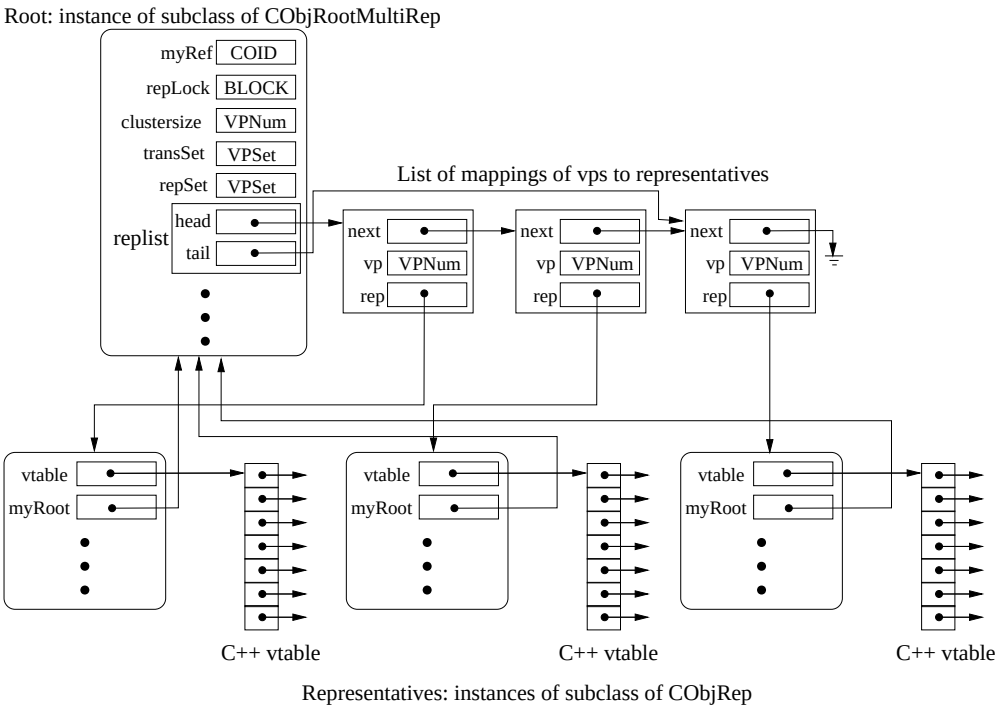


Figure 5.6: A distributed Clustered Object instance created from the standard base Clustered Object Classes. It is composed of multiple Representatives, which are instances of a subclass of CObjRep, and a single instance of root which is derived from CObjRootMultiRep as its root.

made. A critical challenge is developing implementations which are capable of servicing all hot operations of an object in a localized fashion despite potentially conflicting requirements. For example, we have seen that the distributed performance counter was capable of achieving scalable performance for updates. However, if we look at the performance, in cycles, to obtain the counter's value, we see that the costs grow proportionally with the number of processors accessing the counter. In the case of the performance counter, it is clear that the update method is the hot method and having less than scalable performance for obtaining the counter's value is likely acceptable for many applications. However, it is not at all clear that, for more complex objects under varying and potentially conflicting loads, it will be possible to make such simple tradeoffs and still achieve global, stable, scalable performance. This is one of the central questions this work explored in the examples of Chapter 4.

5.4 Clustered Object Protocols

This section discusses the key Clustered Object protocols. We will focus on the support for distributed Clustered Objects, as non-distributed objects are a special case simplification. The fol-

```
class PerfCtrDistributed : public PerfCounter {
    // Root definition for the distributed performance counter
    class PerfCtrDistributedRoot : public CObjRootMultiRep {
        public:
            // All roots must define what reps should be created when
            // the first miss on a processor occurs
            virtual CObjRep * createRep(VPNum vp) {
                CObjRep *rep=(CObjRep *)new PerfCtrDistributed();
                return rep;
            }
        DEFINE_GLOBAL_NEW(PerfCtrDistributedRoot);
    };
    // Per Representative Definitions (local data and methods)
    sval _val;
    PerfCtrDistributed(): _val(0) {}
public:
    // Create method used to instantiate new instances
    static SysStatus Create(PerfCounterCOID coid) {
         coid = (PerfCounterCOID)
               (new PerfCtrDistributedRoot)->getRef());
    }
    // Loop across all the reps and aggregate their individual values
    virtual SysStatus value(sval &count) {
        PerfCounterDistributed *rep=0;

        count=0;
        COGLOBAL(lockReps());
        for (void *curr=COGLOBAL(nextRep(0,(CObjRep *&)rep));
             curr; curr=COGLOBAL(nextRep(curr,(CObjRep *&)rep))) {
             count+=rep->_val;
        }
        COGLOBAL(unlockReps());
        return 1;
    }
    // increment this reps value
    virtual SysStatus inc() { FetchAndAddSignedSynced(&(_val),1); }
    DEFINE_LOCALSTRICT_NEW(PerfCtrDistributed);
};
```

Figure 5.7: Code for a simple distributed performance counter.

lowing is an overview of the Clustered Object protocols:

Allocation and Initialization: protocols which enable various allocation and initialization scenarios.

Inter-Rep: protocols which enable a Representative to identify and cooperate with the other Representatives of a Clustered Object.

Destruction and RCU: protocols for leveraging semi-automatic garbage collection for the internal management of destruction of a Clustered Object and support for Read Copy Update primitives.

Hot-swapping: protocols which enable an instance of a Clustered Object to be replaced dynamically with another type-compatible instance.

We discuss each of these in turn. Hot-swapping is discussed separately in Chapter 6.

5.4.1 Allocation and Initialization

There are two categories of Clustered Object instances accessible in an address space; well-known objects and dynamically allocated objects. The COSMgr provides mechanisms for the address space initialization code to instantiate the well known objects, including the COSMgr itself. The default Clustered Object classes cooperate with the COS via the COSMgr to implement allocation and initialization of the dynamic objects. In this subsection, we will focus on dynamic object allocation and initialization.

In order to ease the programmer's burden, support for lazy Representative mapping and management has been factored out from Representative instantiation and incorporated into the `CObjRootMultiRep` root class. When a root is allocated, the default constructors invoke the `alloc` method of the well-known COSMgr Clustered Object. As discussed earlier, the COSMgr is the primary object which manages the Clustered Object System. The `alloc` method reserves a free entry in the global translation table for a new Clustered Object and stores a pointer to the root of the new Clustered Object in the entry. This implicitly allocates a new Clustered Object identifier (as a reminder, a Clustered Object identifier, or ref, is the address of an entry in the local translation table which corresponds to an entry in the global translation table). The COSMgr manages Clustered Object identifiers by maintaining simple free lists of global table entries. To ensure scalability and facilitate the identification of the processor on which a Clustered Object was allocated (the home processor for

the Clustered Object), the COSMgr manages the global translation table on a per-processor basis. There is one COSMgr Representative for each processor and each manages a distinct range of the global translation table. Given this partitioning, the `alloc` method of a COSMgr Representative need not synchronize or co-ordinate with other Representatives, as each manages a unique range of the global translation tables using independent data structures.[6]

Installing a pointer to the root in the global translation table entry, associated with the newly allocated Clustered Object identifier, ensures that any access to the external interface of the object will result in the miss-handling mechanisms invoking the `handleMiss` method of the installed root. More specifically, the low-level miss-handling code will invoke the static `GenericDefaultFunc` method of the COSMgr. This method translates the identifier of the Clustered Object on which the miss occurred into a pointer to the associated global translation table entry. Each global translation table entry contains a request counter which is used to record all in-flight invocations of the `handleMiss` method on the root. This counter allows the hot-swapping and destruction algorithms, which are discussed in more detail later, to turn away new misses on an object and wait for current in-flight misses to complete. To this end, after translating the Clustered Object identifier to a pointer to the associated global translation table entry, the `GenericDefaultFunc` increments the request counter and then invokes the `handleMiss` method of the associated root.

`GenericDefaultFunc` expects the `handleMiss` method of the root to return back a pointer to a Representative of the Clustered Object. `GenericDefaultFunc` then continues execution by directing the low-level miss-handling code to re-invoke the original method which caused the miss on the Representative returned. Note that the `GenericDefaultFunc` does not install the Representative into the local translation table, rather this is left to the Clustered Object itself. Doing so allows for specialized Clustered Object roots to be developed which do not cache Representatives in the local translation tables but rather simply redirect calls to specified Representatives.

The `CObjRootMultiRep` supports a basic notion of processor clusters, in which a cluster size is specified as an instantiation parameter. Processors are grouped into clusters based on the specified size and their processor number. For example, if a cluster size of four is specified then `CObjRootMultiRep` will treat processors 0-3 as one cluster, 4-7 as another and so on. On NUMA architectures it would be natural to match the cluster size for some objects to the machine's node size.

[6]The use of partitioning is viable as the translation tables are backed by pagable memory and a given processor's range is densely managed.

The `handleMiss` method of the `CObjRootMultiRep` class implements the following cluster-based behaviour for misses. For each cluster of processors, it instantiates a new Representative only on the first miss which occurs on any processor of the cluster. For every miss on a processor the `CObjRootMultiRep` caches a pointer to the Representative for the associated cluster in the processor's local translation table. Additional accesses to the Clustered Object on the processor will be directly serviced by the Representative without a miss. It is important to note, however, that the local translation table is treated only as performance optimization with respect to functionality. The COS semantics dictate that local translation table entries can be flushed at any time. Therefore, the root must serve as the ultimate authority for maintaining the mappings of the Representatives to processors and no code including that of the root should rely on the local translation tables for anything beyond a cache. As such, the `CObjRootMultiRep` has been implemented to maintain these mappings and is able to re-establish the correct translation for any processor independent of whether a miss has or has not previously occurred on the processor.

Roots for distributed Clustered Objects automatically inherit the above behaviour for miss-handling from the `CObjRootMultiRep` root class. A developer need only specify the actions that are required to actually instantiate a new Representative and initialize it, by implementing a `createRep` method which the default methods of `CObjRootMultiRep` invoke as needed.

In order to provide this miss-handling behaviour, the `handleMiss` method of the `CObjRootMultiRep` class maintains a number of logical facts:

- The set of processors on which a translation, in the form of a cached pointer to a Representative, has been established. In the current implementation this is maintained in the `transSet` member (see Figure 5.6).
- The set of processors on which a Representative has been instantiated. In the implementation this is maintained in the `repSet` (see Figure 5.6).
- The mappings of processors to Representatives is maintained by the `replist` member (see Figure 5.6).

These facts are also used to provide the services that allow a developer of a Clustered Object to locate and iterate over the Representatives, as described in the next subsection, and also allow the object to provide the COS with the necessary information to correctly interact with the destruction and hot-swapping protocols. To simplify implementation, each instance of `CObjRootMultiRep` uses a single lock (`repLoc`) to protect the data structures it uses to maintain these facts. The

actual data structures and synchronization methodology are not as important as the facts that the `CObjRootMultiRep` maintains. In the long run we expect the implementation of `CObjRootMultiRep` to be optimized as necessary.

5.4.2 Inter-Rep Protocols

As discussed earlier, using the standard base classes for Clustered Object development ensures that the root of the Clustered Object is accessible from the Representatives via a standard root pointer in the Representative instances (as illustrated in Figure 5.6). The `CObjRootMultiRep` provides the following methods to access the Representatives:

lockReps: This method acquires the `repLock`, stops any misses from occurring and hence ensures that the set of Representatives does not change.

unlockReps: This method releases the `repLock`, re-enabling misses.

nextRep: This method provides a means for iterating over the list of Representatives. If bracketed by calls to `lockReps` and `unlockReps`, then iterations are ensured of covering the exact set of Representatives. If `nextRep` is used without first calling `lockReps`, then it is still safe, but the iteration is not guaranteed to visit Representatives added during the iteration.

findRepOn: Given a processor number, this method returns the rep associated with the cluster that the VP is a part of. If no Rep exists, this method returns NULL.

getRepOnThisVP: This method returns a Rep associated with this vp. If one does not already exist, one is created, added to the Rep list and returned.[7]

These methods are utility methods provided for the developer of a distributed Clustered Object to access the state maintained by default. It is expected that if a developer has special requirements for traversing the Representatives of a Clustered Object, she will explicitly construct the necessary support. For example as in the case of the COSMgr object, the Representatives are explicitly linked together in a ring to support the neighbour structure required. If however, a simple gather is being implemented, as in the case of the `value` method of the `perfCounter` example presented in Figure 5.7, the standard `lockReps`, `unlockReps` and `nextRep` suffice.

[7]This method is named in order to be explicit that it will only return a Rep for the current virtual processor on which the method is invoked.

These methods have been sufficient for simple use and validation of the Clustered Object model for this dissertation. Future work may explore optimizations and support for $O(1)$ translation of processor number to Representative in a standard way. Future work may also explore explicit support for lock free traversal of the Representatives. This would help alleviate the potential for programmers accidentally introducing subtle deadlocks. The methods as implemented in this work must be used with care, since the `lockReps` and `unlockReps` manipulate the lock used to serialize execution of the `missHandling` routine and hence actions bracketed between them cannot be arbitrary. Actions which induce a miss on the object either directly or indirectly can cause a deadlock. In the case of simple gathers of data values, using direct Representative pointers is not a problem.

5.4.3 Destruction and RCU

As many have noted, the construction of dynamic parallel systems software has a number of challenges, one of which is the problem of existence locking [65, 107]. Given the dynamic allocation of data structures which are accessed by concurrent threads, there are subtle race conditions which must be dealt with to ensure that threads do not access stale or dangling references. For example, let us assume that a thread A allocates a data item D to which it receives a pointer from the allocation. Thread A then spawns a new thread B, to which it passes the pointer to D. Thread A then attempts to access D; however if no additional steps are taken, thread B may already have deallocated D and thus rendered A's pointer stale or dangling. The standard techniques employed to deal with such situations are existence locks and reference counters.

Gamsa proposed the use of a semi-automatic garbage collection system that exploits the nature of systems software in order to alleviate some of the burden associated with such race conditions between allocation, access and deallocation of dynamic objects [48, 49]. The approach partitions the references to a data item into two categories: permanent and temporary. Permanent references are those which are stored in memory structures such as tables or objects, whereas temporary references are those which are in registers or on the thread stacks. Gamsa's scheme advocates that all permanent references be explicitly invalidated prior to submitting the object to the garbage collection service. The garbage collection service then ensures that the object is only deallocated when no temporary references exist. At first glance this may not seem significant, as the programmer must still manually deal with the permanent references; however, such a scheme has a number of advantages. Since temporary references are automatically dealt with, all that needs to be done is

to invalidate all references, without regard for order or atomicity. For example, one can simply set all permanent references to a value which will cause future accesses to be denied; e.g., to destroy an object O to which there is a permanent reference in a table, one simply updates the table entry with a reference to a well-known static object which returns an error on all accesses and then submits O to the garbage collector. There is no need to synchronize around the table entry; all future readers of the entry will fail to gain access to the object. Any threads that currently exist, and may have a temporary reference, will be accounted for by the garbage collector.

McKenney et al. coined the term *Read-Copy-Update (RCU)* synchronization to describe a generalization of the above methodology and utilized it in PTX [97] and in Linux to implement a number of optimizations such as lock-free module loading and unloading [93, 95, 133]. K42's Clustered Object system utilizes RCU mechanisms to provide Clustered Object garbage collection as proposed by Gamsa and dynamic replacement or hot-swapping of Clustered Objects [5]. In addition to implementing the K42 garbage collection infrastructure, we have developed a standard set of protocols to facilitate the destruction for Clustered Objects which are composed of multiple Representatives. We proceed by first discussing the garbage collector and RCU in general terms and then describe the supporting K42 mechanisms and Clustered Object protocols.

The garbage collector is responsible for delaying deallocation of an object until all the threads which have a (temporary) reference to the object terminate. In order to make the implementation of such a collector tractable, rather than identifying explicitly which threads have a reference to the object, the collector defers deallocation until all threads that existed at the time the object was submitted have terminated. This is in general insufficient as there is no guarantee that all threads will terminate in a timely fashion, or at all. However, operating systems are event-driven in nature where the events are serviced by short-lived threads. The majority of activity in the OS can be represented and serviced as individual requests with any given request having an identifiable start and end. This nature can be leveraged to enable the garbage collector and RCU synchronization algorithms in general.

By associating changes of system data structures with an epoch of requests, one can identify states in which a data structure is no longer being referenced. For example, to swing the head pointer of a linked list from one chain of nodes to another, one can divide the accesses to the list into two epochs. The first epoch includes all threads in the system that were active before the swing, and the second epoch includes any new threads begun after the swing. Because new threads will only be able to access the new chain of nodes, nodes of the old chain are guaranteed to no

longer be in use once the threads in the first epoch have ended. At this point, the old chain is quiescent and can be modified at will (including being deleted).

The key to leveraging *RCU* techniques is being able to divide the work of the system into short-lived "requests" that have a cheaply identifiable start and end. In non-preemptive systems such as PTX and Linux[8], a number of key points (e.g., system calls and context switches) can be used to identify the start and end of requests. K42 is preemptable and has been designed for the general use of RCU techniques via the Clustered Object garbage collection services. K42's design does not use long-lived system threads nor does it rely on blocking system-level threads. All blocking occurs outside of the system context and data structures by blocking user-level threads and the use of continuation structures. By ensuring that all system requests are handled on short-lived system threads, creation and termination occurs in a timely fashion and can be used to identify the start and end of requests.

Specifically, it is possible to determine when all threads in existence on a processor at a specific instance in time have terminated. This can be achieved by using a generation based thread tracking algorithm, where a generation is identified by a generation number g and associated with a thread counter. For each thread assigned to a generation, the generation's thread counter is incremented and when a thread terminates the thread counter of the generation to which the thread is assigned is decremented. All new threads are assigned to the current generation ($g_{current}$) and at a specified threshold a new generation ($g_{current+1}$) is created and made current (with initially $g_{current} = g_0$). Doing so bounds the number of threads assigned to any generation. To determine when all threads in existence prior to time t have terminated, one records the generation number of the current generation at t ($g_t = g_{current}$) and then forces a new generation to be created and made current. When the sum, $\sum_{i=t}^{0} g_i$, is zero we can assert that all threads in existence prior to t have terminated. An approximation of this algorithm can be efficiently implemented with a fixed number of generation counters. The implementation in K42 use two generation counters.

K42's threads are assigned to one of two generation counters.[9] Each generation counter records the number of threads that are active and assigned to it. At any given time, one of the generation counters is identified as the current generation counter and all new threads are assigned to it. To determine when all the current threads have terminated, the pseudo code presented in Figure 5.8 is used. In K42, the process of switching the current generation counter is called a generation swap.

[8]Schemes for extending the current RCU support in Linux to preemptive versions have been proposed [95].
[9]The design supports an arbitrary fixed number of generation counters but only two are used currently.

```
i=0
while (i<2)
    wait until non-current generation is zero and make it the current generation
    i=i+1
```

Figure 5.8: Pseudo code to determine when current threads have terminated.

Two generation swaps are required to establish that the current set of threads have terminated. It should be noted that the use of a fixed number of generations is an approximation to the generalized algorithm, as there is the potential for new threads to delay the declaration that all prior threads have terminated. We have, however, in practice found this implementation to be timely and accurate even in the face of preemption.

On top of the generation swapping mechanism, we have implemented the notion of a *thread marker*. Thread markers abstract the generation swapping mechanisms, isolating the client code from the details. Two operations are provided on thread markers: `Set` and `updateAndCheck`. A client creates a thread marker and then invokes `Set` when they want to "mark" a point in time relative to which they would like to determine a quiescent state. When a client wants to know if a quiescent state has been reached they invoke `updateAndCheck` on the marker. This method returns either `ACTIVE`, indicating that active threads exist, or `ELAPSED`, indicating that a quiescent state has been achieved and all threads have terminated. We have found the *thread marker* abstraction general enough to implement the Clustered Object garbage collector and the hot-swapping algorithm. It has also enabled others to implement RCU based facilities such as a non-blocking hash table and a specialized resource reclamation facility outside of the Clustered Object system.

In order to determine when all the threads across all processors of an address space have terminated, the Clustered Object System implements a global thread marker facility. A single token is passed from processor to processor. The processors are organized in a ring, and when a processor receives ownership of the token it sets a local thread marker; when the marker has elapsed it passes the token to the next processor in the ring. Thus, when the token has made a complete circuit, all processors of the address space have established a quiescent state, and hence a global quiescent state is achieved. Various optimizations to this algorithm have been designed but are not within the scope of this thesis.

In the remainder of this subsection, we discuss the Clustered Object protocols implemented

by the `COobjRootMultiRep`, `COobjRep` and the COSMgr that implement distributed reclamation of a multi-Representative Clustered Object. The default implementation of the destruction protocol ensures that the Representatives will be deallocated from the processor on which each was allocated. If the developer wishes to deallocate additional resources in a distributed fashion, she simply needs to add the extra deallocations to the standard C++ destructor of the Representative. Additionally, the default implementation ensures that the root of a Clustered Object is only destroyed after destruction of the Representatives has been initiated and, again, the programmer can extend the actions taken by defining a standard C++ destructor for the root.

There are two types of resources associated with a Clustered Object:

1. Memory of the root and Representatives and any associated memory dynamically allocated.

2. Translation table entries: the set of local translation table entries that have been used to cache pointers to the Clustered Object's Representatives and the global translation table entry associated with the Clustered Object.

In order to reclaim these resources, the COSMgr implements a distributed reclamation algorithm in co-operation with the roots of Clustered Objects. Ultimately, the COSMgr relies on the root of a Clustered Object to correctly reclaim its memory, but actively reclaims the translation resources.

To implement this distributed reclamation, the COSMgr utilizes the distributed structure illustrated in Figure 4.19 of Chapter 4. A Clustered Object is submitted to the COSMgr for destruction via the `destroyCO` method, which operates solely on the processor on which it was invoked, only accessing data structures of the associated COSMgr Representative. This method takes the following actions:

1. Marks the global translation entry for the Clustered Object as being destroyed. This ensures that new misses on the Clustered Object will be turned away with an error.

2. Using the request count of in-flight misses, maintained in the global translation entry, `destroyCO` waits for any current misses to complete.

3. The object is then categorized as either only requiring local reclamation with respect to memory and local translation entries or as requiring remote processing. To do this it queries the root of the object for the set of processors on which it has established translations, via the root's `getTransSet` method. Based on this set, the object is placed on either the `stage1LocalList` or the `stage1RemoteList`. In our discussion we will only focus on the

processing of objects which have been accessed on multiple processors and are hence put on the `stage1RemoteList`.

At this point the `destroyCO` method returns to the caller, and with respect to the caller, the object will be destroyed asynchronously at some future point when the COS has determined that it is no longer possible to access the object.

Associated with each Representative of the COSMgr is a worker event which periodically processes the destruction work on each processor. As indicated above, Clustered Objects submitted for destruction are placed on a `stage1` list for later processing. Each entry on a `stage1` list represents the entry of new work into the distributed reclamation system of the COSMgr and will be processed by the worker event. The worker events establish two states associated with each object submitted for destruction:

Memory Quiet: No new accesses to the memory of the object can occur and a quiescent state has been reached on all of the processors which accessed the object and thus memory reclamation can commence.

Id Quiet: When a quiescent state has been reached on all processors of the system, the global translation resources can be reclaimed.

The main goal of the COSMgr is to establish these states and initiate the actions to reclaim the resources. The `CObjRootMultiRep` class provides a default implementation of a root which both provides necessary information to the COSMgr as well as utilizing the COSMgr services to implement the memory reclamation.

Prior to submitting the object for destruction, it is the programmer's responsibility to ensure that no new accesses to the object can occur via permanent references (see Section 5.1). Therefore, the **Memory Quiet** state can be established by waiting for all threads on each of the processors which have accessed the object to terminate. The COSMgr has been implemented to determine the **Memory Quiet** state in a distributed fashion using a token-based scheme which avoids broadcasting interprocessor interrupts for every object destroyed and amortizing the costs of communication through batching.

At any given time, only one COSMgr Representative is identified as owning the token. When the worker event associated with the Representative which owns the token executes, it waits for all the current threads on the local processor to terminate via the RCU mechanisms and then processes the objects on its `stage1` list. The following steps are taken for each object:

1. Make a copy of the set of processors on which the object has been accessed by querying the root of the object via its `getTransSet` method. The `CObjRootMultiRep` implementation of the root ensures that the set returned is accurate.

2. If the object has been accessed on the processor of the COSMgr Representative associated with the worker event then:

   (a) Reclaims the local translation table entry associated with the object on this processor.

   (b) Takes this processor out of the set of accessing processors.

3. The pointer to the root of the object, along with the potentially modified set of processors which have accessed the object, are copied onto the `transferArray` associated with this COSMgr Representative for further processing.

Similarly, the worker event processes the objects that are on the `transferArray` of the Representative which had the token before. For each object:

1. If the object has been accessed on the processor of the COSMgr Representative associated with the worker event then:

   (a) the local translation table entry associated with the object is reclaimed,

   (b) the processor is taken out of the set of accessing processors, and

   (c) if the set of accessing processors is empty then the object has been processed by all processors which accessed the object. In this case the `cleanup` method of the root is invoked in order to begin memory reclamation.

2. If the set of accessing processors is not empty, then the object is placed on the `transferArray` of this COSMgr Representative for further processing.

After all the objects have been processed, the worker event passes the token to the next COSMgr Representative in the ring. An object submitted for destruction will be progressively processed by each processor that has accessed it, as the token circulates around the ring of COSMgr Representatives. To review: as each COSMgr Representative processes an object, it checks to see if it is the last processor in the set of accessing processors, and if so, it initiates the memory reclamation of the object by invoking the `cleanup` method of the root of the object.

Before looking at how the memory is actually reclaimed, let us examine how the **Id Quiet** state is established. As an object is circulating along with the token, between the Representatives of

the COSMgr, it will eventually be processed by the processor on which the object was allocated[10]. When this processor processes the object, it records the Clustered Object identifier associated with the object on a list of identifiers to be reclaimed the next time it receives the token. Each time a COSMgr Representative receives the token, it reclaims any Clustered Object identifiers on its reclaim list. As discussed earlier, the token's arrival is indicative of a quiescent state having occurred on all processors, and therefore, for any Clustered Object identifier on a COSMgr Representative's reclaim list, the token arrival represents the **Id Quiet** state. Since the local translation entries are reclaimed as the token visits each of the processors on which the object has been accessed, the COSMgr Representative only reclaims the global translation entry as it processes the entries of its Clustered Object identifier reclaim list.

In order to facilitate distributed reclamation of the memory associated with a Clustered Object, the COSMgr maintains an additional per-Representative list of objects, called the `vpCleanupList`. call-backs to the `cleanup` method are periodically made to the roots of the objects on the `vpCleanupList`. When an object is being initially queried for its translation set it is also queried for the set of processors on which it would like to have cleanup call-backs invoked (the vpCleanupSet). This set is circulated along with the object as the token is passed. Each processor specified in the vpCleanupSet adds the object to its `vpCleanupList` when it processes the object. Each time the worker event associated with a COSMgr Representative executes, it invokes the `cleanup` method on the roots of the objects specified on the `vpCleanupList`.

The `CObjRootMultiRep` uses this feature to implement distributed reclamation of its Representatives. Its `getVPCleanupSet` specifies that it would like to have `cleanup` call-backs invoked on the set of processors on which its Representatives were instantiated. This establishes periodic invocations of the `cleanup` method of the root for all processors for which a Representative was instantiated. By default, these invocations do not take any action but simply return a value that indicates that invocations should continue to occur and are kept on the `vpCleanupList` of the associated COSMgr Representative. However, once the **Memory Quiet** state has been established for the object, the COSMgr executes a single invocation of the `cleanup` method of the object, passing it a unique parameter indicating that the object should now begin memory reclamation. On this invocation the `CObjRootMultiRep` changes the behaviour of the call-backs to indicate that they should now reclaim the resources. Future call-backs now destroy the Representatives associ-

[10]The token passing memory reclamation described in the previous paragraph is implemented to continue to pass the object around if it has not visited the allocating processor.

ated with the processors. Additionally, they atomically check to see if they are destroying the last Representative and if so the root object is also destroyed.

Although the above may seem complex, the default semantic that the COSMgr and `CObjRootMultiRep` provide is straightforward and simple to use:

- The standard C++ destructor for the Representative instances will automatically be invoked when it is safe to do so, on the processor on which the Representative was allocated, in a parallel fashion. Thus the programmer can simply place any per-processor destruction logic in the standard C++ destructor of the Representative. If none is specified, the default action will be to deallocate the memory of the Representative.

- The standard C++ destructor for the root will be invoked as the final action of the object, only after the cleanup method has been invoked on all of the Representatives. Thus all global resources can be destroyed in the standard C++ destructor for the root.

Additionally, an advanced programmer can over-ride the default behavior and implement a completely custom reclamation behavior, without modification to the Clustered Object system itself by over-riding the methods of the `CObjRootMultiRep` class.

5.5 Summary

In this chapter we have described the Clustered Object infrastructure, which provides a simple but flexible model for Clustered Object development. The infrastructure provides the developer with a structure in which every Clustered Object is composed of two types of components; a single Root and one or more Representatives. Standard protocols for the construction, management and destruction of a Clustered Object are implemented by reusable components of the infrastructure, easing the programmers' burden. A runtime Clustered Object System provides basic allocation and garbage collection facilities. These services are implemented via a distributed Clustered Object Manager.

The support for Clustered Objects described here has enabled the examples presented in Chapter 4 and provides the basic object model for K42. All objects in K42 are Clustered Objects and over the span of the development of K42, many developers have created new Clustered Objects, even if they were not aware of it. Portions of my Clustered Object infrastructure has been in use since 1998.

Chapter 6

Hot-swapping

Distributed implementations can offer better scalability, but they also typically suffer greater overheads when scalability is not required. Distributed implementations also tend to optimize certain operations, improving their scalability, while increasing costs of other operations. In order to provide a means for coping with the tradeoffs of using distributed implementations, we have developed a technique for dynamically replacing a live Clustered Object instance with a different, but compatible, instance. This mechanism can be used to switch between shared and distributed implementations and additionally enable other forms of dynamic adaptation. Our contribution has been in the development of the mechanisms and the algorithm for the dynamic replacement of a Clustered Object instance.

There are a number of challenges in the design of a hot-swapping infrastructure capable of dynamically switching a "live" software component: 1) avoid adding overhead to normal method invocations, 2) avoid complicating the design of those objects that can be switched, 3) ensure the switching code is scalable, 4) correctly handle in-flight requests to the object being switched, 5) avoid deadlock during the switch, and 6) guarantee integrity when transferring state from the old to the new object instance. The distributed nature of Clustered Objects further exacerbates these challenges as it can mean having to swap the multiple constituents of a component across multiple processors in a coordinated way.

In this chapter, we describe the basic hot-swapping mechanism and algorithm as constructed on top of the Clustered Object infrastructure. We begin with an overview and then present the details.

6.1 Overview

Our swapping mechanism allows any Clustered Object instance to be hot-swapped with any other Clustered Object instance that implements the same interface. Moreover, swapping is transparent to the clients of the component and thus no support or code changes are needed in the clients.

6.1.1 Algorithm Overview

The outline for our hot-swapping algorithm is as follows (and is described in more detail further below):

(*i*) instantiate the replacement Clustered Object instance;

(*ii*) establish a quiescent state for the instance to be replaced (potentially blocking calls as necessary) so that we know it is no longer being used;

(*iii*) transfer state from the old instance to the new instance;

(*iv*) swap the new instance for the old, adjusting all references to the instance;

(*v*) unblock calls that were blocked (which now will be serviced by the new instance); and

(*vi*) deallocate the old instance.

There are three key issues that need to be addressed in this design. The first and most challenging issue is how to establish a quiescent state so that it is safe to transfer state and swap references. The swap can only be done when the instance state is not currently being accessed by any thread in the system. Perhaps the most straightforward way to achieve a quiescent state would be to require all clients of the Clustered Object instance to acquire a reader-writer lock in read mode before any call to the object (as is done in the re-plugging mechanism described by McNamee et al. [98]). Acquiring this external lock in write mode would then establish that the object is safe for swapping. However, this approach adds overhead for the common case and can cause locality problems, defeating the scalability advantages of Clustered Objects. Further, the lock could not be part of the component itself and the calling code would require changes. Our solution avoids these problems by leveraging the same basic RCU mechanisms provided by the Clustered Object System as discussed in Section 5.4.3.

The second issue is deciding what state needs to be transferred and how to transfer the state from the old component to the new one, both safely and efficiently. We provide a protocol that

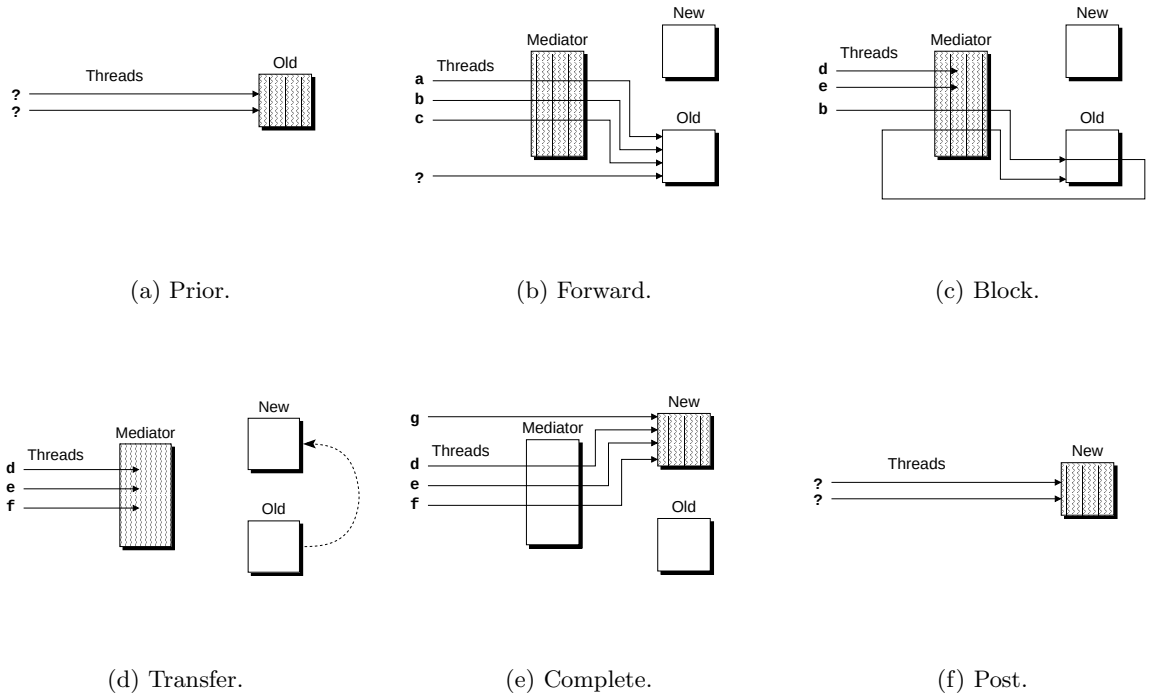
Clustered Objects developers can use to negotiate and carry out the state transfer. Although the state could be converted to some canonical, serialized form, one would like to preserve as much context as possible during the switch, and handle the transfer efficiently.

The final issue is how to swap all of the references held by the clients of the component so that the references point to the new instance. In a system built using a fully-typed language such as Java, this could be done using the same infrastructure as is used by garbage collection systems. However, this would be prohibitively expensive for a single component switch and would be overly restrictive in terms of systems language choice. An alternative would be to partition a hot-swappable component into a front-end component and a back-end component, where the front-end component is referenced (and invoked) by the component clients and is used only to forward requests to the back-end component. Then there would be only a single reference (in the front-end component) to the back-end component that would need to be changed when a component is swapped, but this adds extra overhead to the common call path. Given that all accesses to a Clustered Object in K42 already go through a level of indirection, namely the Local Translation Table, the more natural way to swap references in our system is to overwrite the entry pointers in a coordinated fashion.

6.2 Details

To implement the swapping algorithm outlined above, a specialized Clustered Object called the Mediator Object is used during the swap. It coordinates the switch between the old and new objects, leveraging the Clustered Object infrastructure to implement the swapping algorithm. To handle the swapping of distributed Clustered Object instances with many parallel threads accessing it, the Mediator is itself a distributed Clustered Object that implements the swapping algorithm in a distributed manner by utilizing a set of worker threads.

The Mediator establishes a worker thread and Rep on each processor for which the original Clustered Object instance has been accessed. The Mediator instance is interposed in front of the target Clustered Object instance and intercepts all calls to the original object for the duration of the swapping operation. The details of how the interposition is achieved will be described later. The worker threads and Mediator Reps transit through a sequence of phases in order to coordinate the swap between the old Clustered Object instance and the new one replacing it. The Mediator Reps function independently, only synchronizing when necessary in order to accomplish



(a) Prior. (b) Forward. (c) Block.

(d) Transfer. (e) Complete. (f) Post.

Figure 6.1: **Component hot-swapping.** This figure shows the phases of hot-swapping with respect to a single processor and the Reps: prior, forward, block, transfer, complete, and post. In the forward phase, new calls are tracked and forwarded while the system waits for untracked calls to complete. Although this phase must wait for all old threads in the system to complete, all threads are allowed to make forward progress. In the block phase, new calls are blocked while the system waits for the tracked calls to complete. By blocking only tracked calls into the component, this phase minimizes the blocking time. In the transfer phase, all calls to the component have been blocked, and state transfer can take place. Once the transfer is complete, the blocked threads can proceed to the new component and the old component can be garbage collected.

the swap. Figure 6.1 illustrates the phases that a Mediator Rep goes through while swapping a Clustered Object. The remainder of this section describes how the Mediator Reps and worker threads accomplish the swap. We present the actions that occur on a single processor but in the general case these actions proceed in parallel on multiple processors.

Prior to initiating the swap (Figure 6.1a), the old object's Reps are invoked as normal. The first step of hot-swapping is to instantiate the new Clustered Object instance, specifying that it not be assigned a COID, and that the installation of its root into the Global Translation Table be skipped. Second, a new Mediator instance is created and passed both the COID of the old instance and a pointer to the Root of the new instance. The Mediator then proceeds to interpose itself in front of the old instance.

Interposing a Mediator instance in front of the old Clustered Object instance ensures that future calls temporarily go through the Mediator. To accomplish this, the Mediator instance must override both the Global Translation Table entry root pointer and all the active Local Translation Table entries' Rep pointers. To change the Global Translation Table entry Root pointer, it must

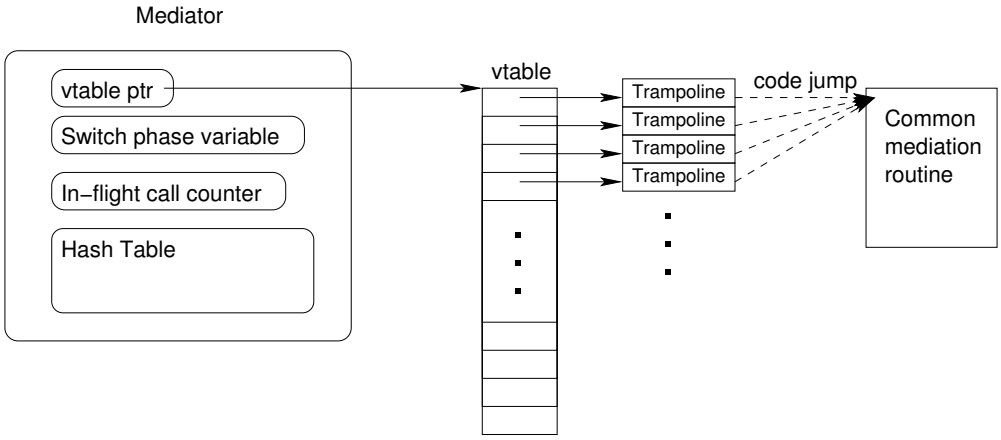


Figure 6.2: Mediator Rep implementation

ensure that no misses to the old object are in progress. As part of the standard Miss-Handling infrastructure, there is a reader-writer lock associated with each Global Translation Table entry and all misses to the entry acquire this lock in read mode. In order to atomically change the Global Translation pointer, the associated reader-writer lock is acquired for write access, ensuring that no misses are in progress. When the lock has been successfully acquired, the Root pointer of the entry is changed to point to the Root of the Mediator and all future misses will be directed to it. The Mediator remembers the old object's Root in order to communicate with it. During this process there may be calls that are in flight to the old Clustered Object, and they proceed normally.

Swinging the Root is not sufficient to direct all calls to the Mediator instance. This is because some Rep pointers may already be established in the Local Translation Table entry associated with the old instance, causing some calls to proceed directly to the Reps of the old instance. To handle this, the Mediator spawns a worker thread on all the processors that have accessed the old object. These threads have a number of responsibilities, but their first action is to reset the Local Translation entry on each processor back to the Default Object. This ensures that future accesses will be directed to the Mediator Object via the standard Miss-Handling process. Because the Root maintains the set of processors for which it has suffered a miss, the Mediator can query the old object's Root to determine for which processors to spawn threads.

On each Mediator miss, the Mediator Root installs a new Mediator Rep into the Local Translation Table for the processor on which the miss occurred. The Mediator Reps are specialized C++ objects similar to the Default Object. They are designed to handle hot-swapping of any Clustered Object transparently. To do so, the Mediator Rep intercepts all calls and takes action based on the current phase of the Rep (Figure 6.1).

Figure 6.1, parts b, c, d and e, illustrate a single Mediator Rep in the different phases of a

swap (described in more detail below). Once the Mediator Rep has been installed into the Local Translation Table entry, virtual method calls that would normally call one of the functions in the original object end up calling the corresponding method in the mediator. A small amount of assembly glue captures the low-level state of the call, including the parameter passing registers and return address. The actions that the Mediator Rep has to take on calls during the various phases of swapping include: forwarding the call to the old or new instance depending on the phase and keeping a count of active calls (increment the counter prior to forwarding the call and decrement it after the forwarded call returns), selectively blocking calls, and releasing previously blocked calls. To be transparent to the clients and the target Rep when the call is being forwarded, the Mediator Rep may not alter the stack layout and hence it must only use Rep-local storage to achieve the appropriate actions. As can be seen in Figure 6.2, the Mediator Rep utilizes three other data members other than its vtable pointer.

The vtable of the Mediator Rep, like that of the Default Object, is constructed to direct every call regardless of its signature to a single common mediation routine. When a phase requires that new calls be tracked and forwarded, the Mediator Rep uses an in-flight call counter to track the number of live calls. Because the counter needs to be decremented when the call completes, the Mediator must ensure that the forwarded call returns to the mediation routine prior to returning to the original caller. This means that the Mediator Rep must keep track of the point to which to return after decrementing its in-flight call counter on a per-thread basis. To maintain transparency it avoids using the stack, in the current, version by maintaining a hash table indexed by thread id to record the return address for a given thread. The Mediator Rep also uses a data member to track its current phase. The phases are detailed in the following paragraphs.

6.2.1 *Forward*

This initial phase is illustrated in Figure 6.1b. The Mediator stays in this phase until it determines that a quiescent state has been reached, with respect to the threads started prior to the swap being initiated. Specifically the Mediator determines that there are no longer any threads that were started prior to the swap initiation still accessing the object. To detect this, the worker thread utilizes thread marker services of K42's Read-Copy-Update mechanism.(See Section 5.4.3 for details.)

By waiting for all threads that were in existence when the swap was initiated to terminate, we are sure that all threads accessing the old object have terminated. However, to ensure system

responsiveness while waiting for these threads to terminate, new calls to the object are tracked and forwarded to the old instance by the Mediator Rep using its in-flight call counter and hash table. The thread descriptors are also marked as being in a forwarded call in order to simplify deadlock avoidance as described in the next paragraph. Such calls are referred to as tracked calls. Once the worker thread, using the generation mechanism, determines that all the untracked threads have terminated, the Mediator Rep switches to the *Block* phase. Note that the *Forward* phase, and transition to the *Block* phase, happen independently and in parallel on each processor, and no synchronization across processors is required.

6.2.2 *Block*

In this phase, the Mediator establishes a quiescent state, guaranteeing no threads are accessing the old Clustered Object on any processor. To do this, each Mediator Rep establishes a quiescent state on its processor by blocking all new calls while waiting for any remaining tracked calls to complete. However, because a tracked call currently accessing the object might itself call a method of the object again, care must be taken not to cause a deadlock by blocking any tracked calls. This is achieved by checking the thread descriptor to determine if the thread is in a tracked call. This also ensures that concurrent swaps of multiple Clustered Objects do not deadlock. If a thread in a tracked call, in one object, calls another object that is in the blocked phase it will be forwarded rather than blocked, thus avoiding the potential for inter-object deadlocks. To ensure a quiescent state across all processors, the worker threads must synchronize at this point prior to proceeding. A shared data member in the Mediator root is used for this purpose.

6.2.3 *Transfer*

Once the Blocked phase has completed, the Transfer phase begins. In this phase the worker threads are used to export the state of the old object and import it into the new object. To assist state transfer, a *transfer negotiation protocol* is provided. For each set of functionally compatible components, there must be a set of state transfer protocols that form the union of all possible state transfers between these components. For each component, the developers create a prioritized list of the state transfer protocols that the component supports. For example, it may be best to pass internal structures by memory reference, rather than marshaling the entire structure; however, both components must understand the same structure for this to be possible. Before initiating a transfer, a list is obtained from both the old and new component instances. The most desirable

format, based on the two lists, is recorded by the Mediator instance. The actual data transfer is carried out in parallel by the worker threads. The worker threads request the state from the old object in the format that was recorded in the Mediator instance and pass it to the new object.

6.2.4 *Complete*

After the state transfer, the worker threads again synchronize so that one may safely swing the Global Translation Table entry to the Root of the new Clustered Object. All the worker threads then cease call interception by storing a pointer to the new Clustered Object's Reps into the Local Translation Table entries so that future calls go directly to the new Clustered Object. The worker threads then resume all threads that were suspended during the *Block* phase and forward them to the new Object (Figure 6.1e). The worker threads deallocate the original object and the mediator and then terminate.

6.3 Summary

In this chapter we have presented a methodology that we have implemented which permits a Clustered Object implementation, with potentially distributed state, to be dynamically switched with another implementation. Given the level of indirection that our Clustered Object infrastructure introduces, one may assume that hot-swapping is straight forward. There are however, subtle issues that must be resolved to enable hot-swapping, including, the detection and establishment of a quiescent state, and how to coordinate the swap of the distributed representatives of a Clustered Object. By leveraging the flexibility of the Clustered Object infrastructure, we were able to construct a specialized Mediator Clustered Object, which can be interposed in front of other Clustered Objects and carry out the hot-swap algorithm.

Chapter 7

Performance

In this chapter we present results of experiments running standard workload benchmarks on K42. The purpose of these experiments is to validate and gain insight into the effectiveness of K42's fundamental architecture in general and the effectiveness of the distributed implementations of key memory management objects in particular. The benchmarks considered were:

SDET: SPEC Development Environment Test, a multiuser workload in which concurrent users issuing standard commands is simulated [125].

Parallel Postmark: Multiple concurrent instances of the Postmark benchmark in which each instance simulates the file accesses of a transactional server such as an email, netnews or a web transaction server [68].

Parallel Make: Multiple concurrent multi-file compilations of an application source tree.

For comparison purposes, and as a reference point, we present results of the same experiments executed on Linux running on the same hardware. This was done primarily to show that the uniprocessor K42 performance is comparable to Linux performance (in part to demonstrate that the scalability results were not achieved by artificially slowing down uniprocessor performance). The experiments were conducted at a specific point in K42 development[1] prior to the completion of K42's Linux compatibility support. Given the constraints of the time we had to make minor modifications to the benchmarks to accommodate K42's functional limitations. The same modified versions of the benchmarks were run on both Linux and K42.

[1]The results were gathered in March of 2003.

While the results presented in this chapter are limited in comparison to what might be possible in the future, we believe the scalability of the fundamental architecture is well represented. All the results presented in this chapter were obtained by running K42 or Linux 2.4.19 as distributed by SuSE (with the O(1) scheduler patch) on PowerPC hardware. We used an S85 Enterprise Server IBM RS/6000 PowerPC bus-based cache-coherent multi-processor with 24 600MHZ RS64-IV processors and 16GB of main memory. For some experiments, we also used a 270 Workstation with four 375MHZ Power3 processors and 512MB of main memory. Unless otherwise specified, all results are from the S85 Enterprise Server. At the time of experimentation, disk IO was limited, so each of the experiments was run using RamFS. Although we have eliminated physical IO we continue to exercise all OS paths. For each main workload experiment, we ran the same script on both K42 and Linux version 2.4.19

Figures 7.1 to 7.3 depict the scalability of K42 with three different benchmarks running on a 24-way shared memory multi-processor. For both the SDET and Parallel Make benchmarks, K42 scales well up to 24 processors. For the PostMark benchmark, as presented, the speedup on K42 is good, and better than on Linux, but does begin to flatten and have variability past 10 processors. Further investigation revealed limitations in our physical memory allocation, which has since been addressed. However, it was not possible to reproduce the PostMark results to validate the improvement. The following sections describe the benchmarks in more detail.

7.1 SDET

The SPEC Software Development Environment Throughput (SDET) benchmark [125] consists of a script that executes a series common Unix commands and programs including ls, nroff, gcc, grep, etc. Each of these are run in sequence. For our experiments, the SDET benchmark was modified by removing some system utilities such as “ps” and “df” which at the time of the experimentation were not supported on K42. To examine scalability, we ran one script per processor. Each script is independent, there is no application-level sharing between scripts, and all data files operated on are unique to a single script. All the user programs (bash, gcc, ls, etc.) are the exact same binary, whether run on K42 or Linux. The same version of glibc 2.2.5 was used, but modified on K42 to intercept and direct the system calls to the K42 implementations.

Figure 7.1 illustrates the speedup obtained when running SDET on Linux and K42, using the original non-distributed VMM Objects and the distributed VMM Objects that were presented in

Chapter 4. We calculate speedup as the improvement in throughput relative to the K42 uniprocessor throughput result, where the throughput numbers are those reported by the SDET benchmark and represent the number of scripts per hour that are executed. For each configuration of processors, we ran the benchmark several times and plot the average as well as the minimum and maximum speedup observed, as indicated by the range bars.

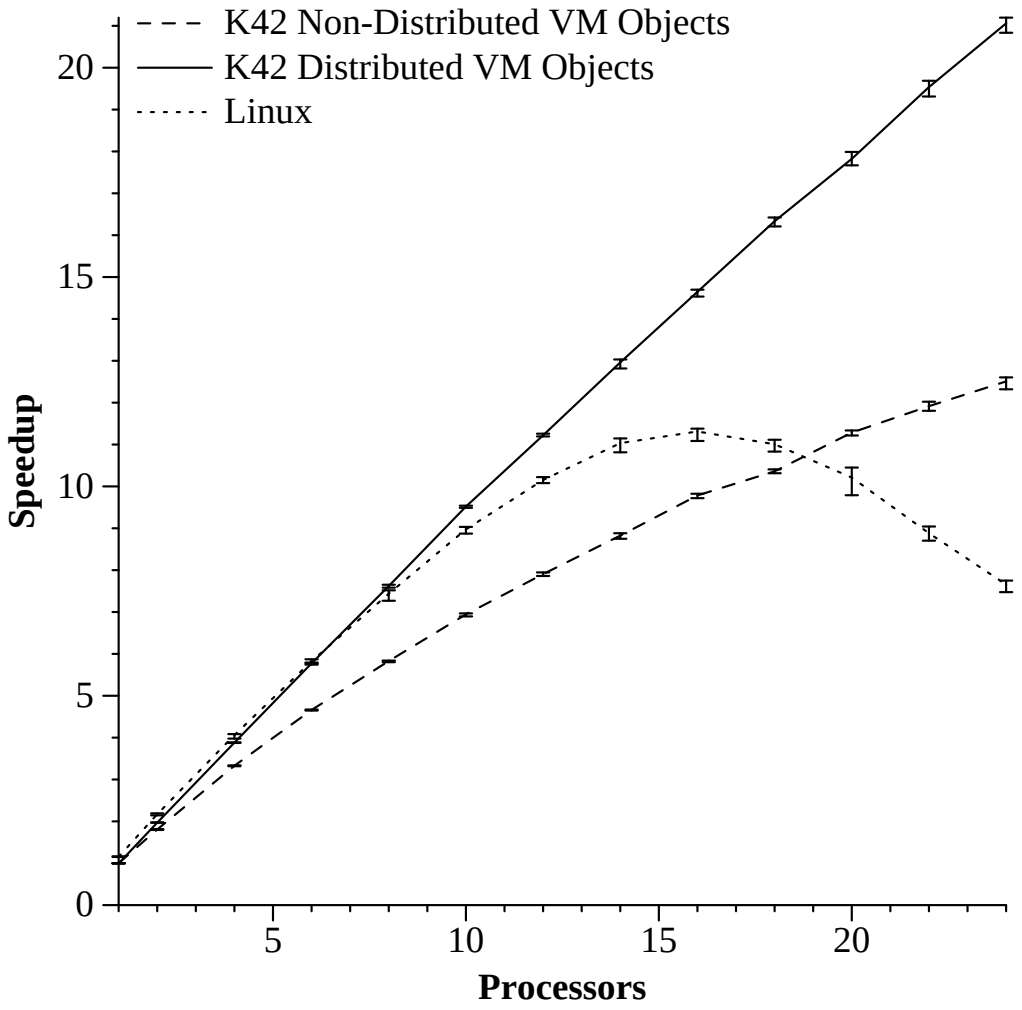


Figure 7.1: Speedup as defined as the increase in throughput of SDET benchmark run on K42 using both the non-distributed and distributed VMM objects and on Linux. Results are relative to the uniprocessor K42 non-distributed VMM result.

7.2 Postmark

Postmark was designed to model the filesystem activity of a combination of electronic mail, netnews, and web-based commerce transactions [68]. It creates a large number of small, randomly-sized files and performs a specified number of transactions on them. Each transaction consists of a randomly

chosen pairing of file creation or deletion with file read or append. A separate instance of Postmark was launched for each processor with corresponding separate directories. Each instance runs independently, and there is no application-level sharing between instances. We ran each instance of the benchmark with 20,000 files, 100,000 transactions, and default options, but disabled Unix buffered files due to K42 limitations at the time of experimentation. The completion time for each instance was recorded. The experiment was repeated several times for each processor configuration and the average time to complete an instance for a processor configuration was calculated, the minimum and maximum completion time was also recorded. The results, calculated as speedup and normalized to the K42 uniprocessor result, are plotted in Figure 7.2. The minimum and maximum speedup for each processor configuration are shown as range bars. The K42 results were obtained with the distributed virtual memory objects. K42's uniprocessor result is approximately two times slower than the Linux uniprocessor result. Postmark is predominately a filesystem benchmark and K42's filesystem support has received little attention with respect to performance tuning, and as such its base costs have not been optimized.

7.3 Parallel Make

Our parallel make experiment is designed to model the common task of several developers building an application in parallel. (In this case the arbitrarily chosen application is GNU Flex). We created one build directory per processor and invoked one sequential `make` in each of these directories in parallel (all `make`'s built from a common source tree, to unique build directories). Each `make` instance runs independently with respect to the output files generated, they do however, share common input files. A driver application synchronized the invocations of each `make` process to ensure simultaneous start times, tracked the run-time of each `make` and reported the final result as an average of all `make` run-times. GNU Make 3.79 and GCC 3.2.2 were used. The results, calculated as speedup and normalized to the K42 uniprocessor result, are illustrated in Figure 7.3 with minimum and maximum speedup for each processor configuration shown as range bars.

7.4 Discussion of Workload Experiments

The distributed implementations discussed in this thesis, although limited to the case study objects of the virtual memory system, are shown to improve the scalability of K42 on the SDET workload and reasonable performance is obtained for the two other workloads. The use of an object oriented

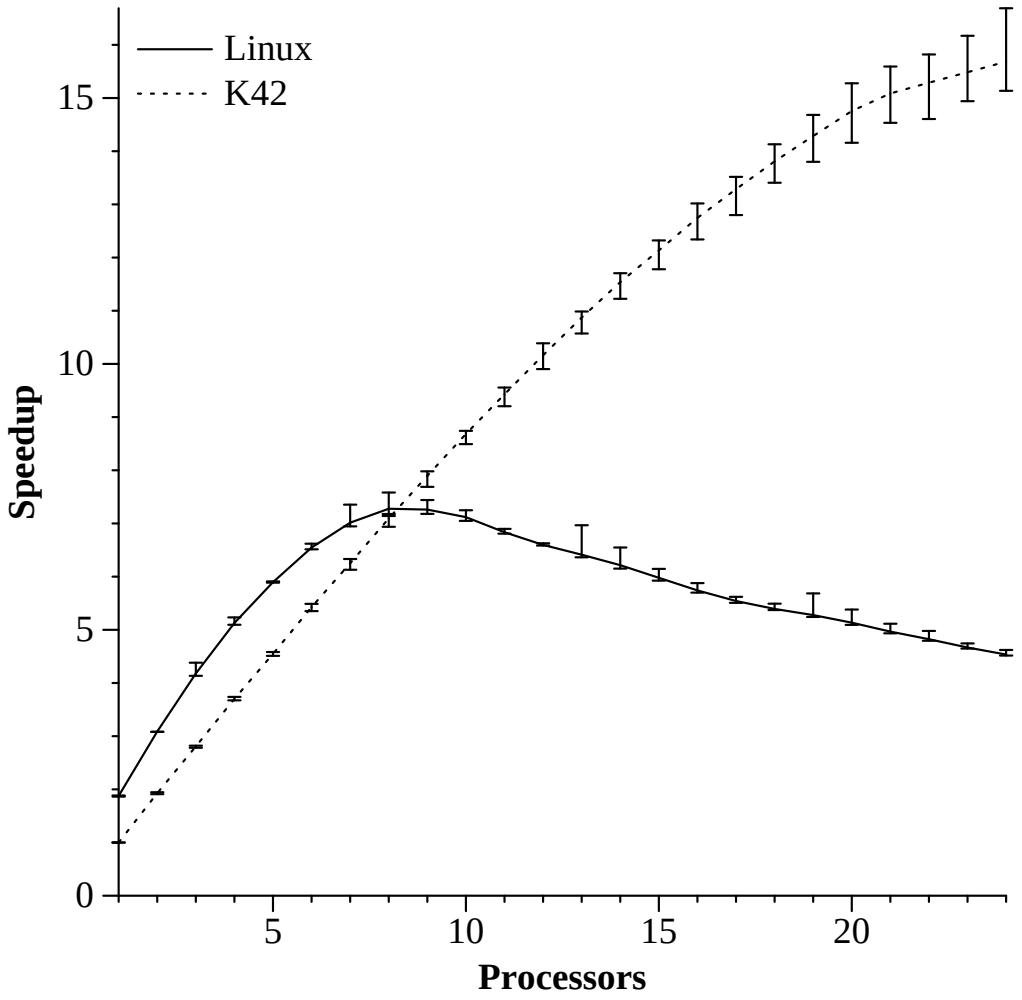


Figure 7.2: Speedup of p independent instances of Postmark normalized to K42 uniprocessor result.

decomposition was not enough to obtain good scalability for the general purpose multi-user workload, in which intuitively good scalability is expected. Shared system object instances can lead to bottlenecks which limit the scalability of the workload.

In our development process, we have used lock contention as our primary guide to identify sharing within the resident page fault path. Having started with non-distributed implementations, with a single lock per-object, the use of standard lock contention analysis identified which object instances on the page fault path were suffering contention. Each contended object implied shared access and hence became a candidate for an implementation optimized to improve locality. As can be observed, the distributed Clustered Object implementations, designed to improve locality on a per-object basis, have helped improve scalability for the system-wide workloads. Taking this approach has allowed us to identify opportunities for locality optimizations without the need

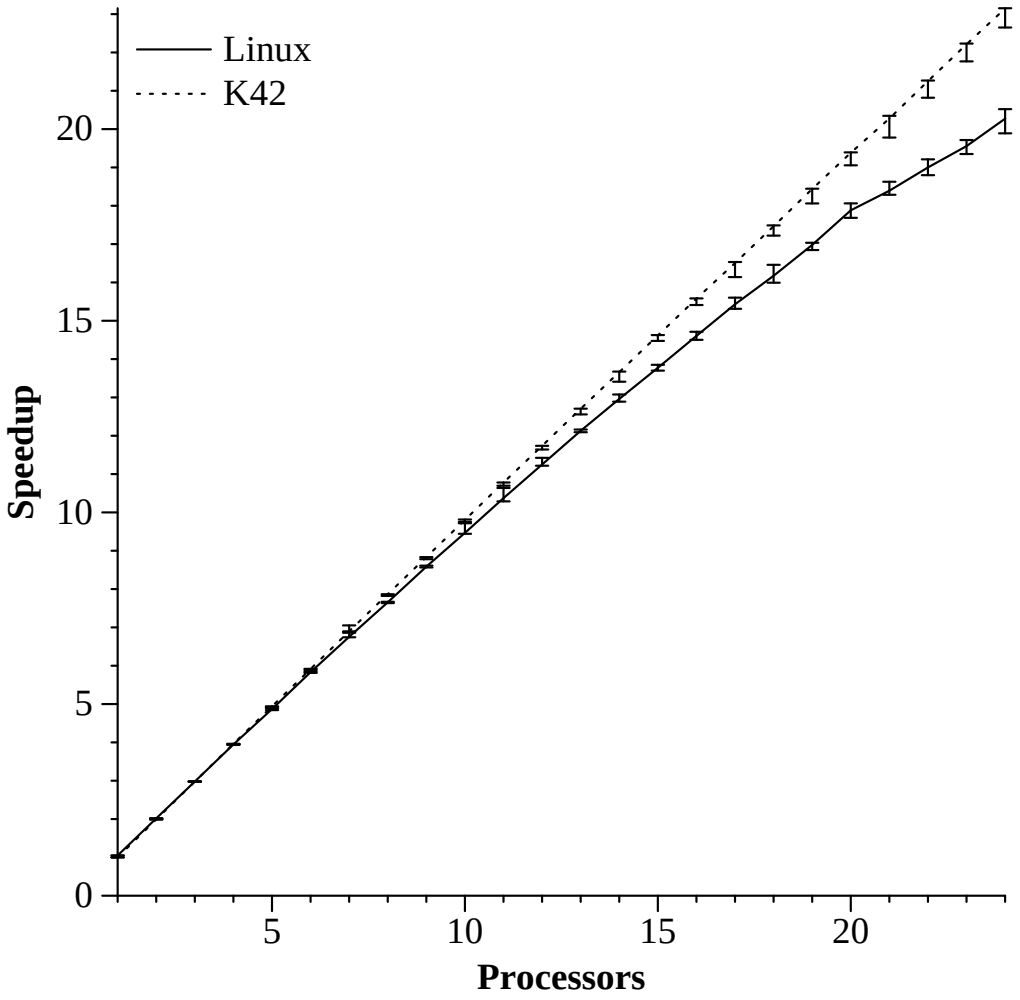


Figure 7.3: Speedup of p independent instances of Parallel Make normalized to K42 uniprocessor result.

for specialized tools to measure hardware contention. Unfortunately, not having such tools also means that we must rely on code path analysis and indirect measurements to validate that the new distributed implementations do not rely on shared memory access.

Like others we have observed that system developers cope with inherent complexity of critical system paths, such as the page fault path, by first exploring the design and implementation using simple large grain synchronization. Improved parallelism is left for a later optimization effort. The Clustered Object approach successfully enabled this natural iteration within the context of an object oriented decomposed path. Scalability optimization of individual objects in isolation did permit improved workload scalability, while preserving the object boundaries, configuration and protocols.

We claim, based on the performance measurements and our analysis, that the distributed im-

plementation have resulted in page fault paths which have good locality and thus have the potential for good scalability for a large range of system configurations. The distributed implementations address not only the lock contention associated with the centralized implementations, but also the shared memory accesses along the paths. The complete validation of this claim is beyond the scope of this work as it requires larger-scale hardware than we have access to and more advanced tools for gathering and analyzing SMP hardware performance.

In general, when we consider the results with respect to Linux, we observe that Linux scales reasonably well up to a certain number of processors, but its performance then begins to degrade for all but the Parallel Make benchmark. Linux scalability will likely continue to improve given the large number of contributions made each year. We believe that new structures and techniques, such as those of K42, will become increasingly important as the number of processors increases further and as newer hardware with increasing processor-memory speed disparity become more common. This trend is evident in the work of Bryant et al. who attempt to apply some similar optimizations to the Linux VMM in an ad hoc manner [19].

7.5 Hot-swapping

In this section we present some initial results evaluating the hot-swapping mechanisms and illustrate some of the performance scenarios that are being explored.

7.5.1 Basic Overheads

In K42, the basic overhead of a Clustered Object invocation consist of the costs associated with the object translation table and C++ virtual function table. The overhead of the object translation table is a single memory load. The overhead of dispatching a virtual function call is approximately 10 cycles. The remainder of the basic overheads for hot-swapping are associated with the Mediator and come into play only when an object is being swapped. There are three performance costs for the Mediator: attaching the Mediator, calling through the Mediator (as opposed to instrumenting the component directly), and detaching the Mediator. Table 7.1 lists these costs as measured on a 270 Workstation. Attaching the Mediator is the most expensive operation, involving memory allocation and object initialization; however, at no point during the attach are incoming calls blocked. Although during detaching the Mediator only requires updating the object translation table, the tear down of the Mediator is listed as an overhead for the process performing the detach.

Operation	μseconds
Attach	17.84 (0.16)
Calling Through	1.40 (0.02)
Detach	4.23 (0.49)

Table 7.1: Basic Mediator overheads. The average cost of attachment, calling through, and detachment of a mediator is listed in microseconds along with its standard deviation.

7.5.2 Hot-swapping Overhead

To determine the expected overhead of hot-swapping, we perform a "null" hot-swap of a contended FCM (swapping the FCM with itself) while running a 4-way SDET on a 270 workstation. Contention on an FCM is detected by many threads accessing an FCM concurrently, such that the lock associated with the non-distributed implementation cannot be acquired due to another thread holding the lock. Although contention is the worst time to swap (because threads are likely to block, increasing the duration of the mediation phases), it is important to understand this sort of "worst-case" swapping scenario. The results are shown in Figure 7.4.

During a single run of 4-way SDET, the system detected 424 points of contention. The average time to perform a hot-swap at these points was 27.6 μs with a 19.8 μs standard deviation, a 1.06 μs minimum and a 132.6 μs maximum. Hot-swapping while the FCM is not under contention is more efficient, because the forward and block phases are shorter. Performing random null hot-swaps throughout a 4-way SDET run gave an average hot-swap time of 10.8 μs. Because most of the time spent doing a hot-swap is spent waiting for the generation to advance (during which no threads are blocked) the effect of these hot-swaps on the throughput of SDET is within normal performance perturbation[2].

7.5.3 Application of Hot-swapping

In this subsection, we present an example use of the hot-swapping mechanism to implement per-file adaptation based on file access. The purpose is to illustrate how the mechanisms we have implemented can be used to select between multiple implementations of an object at run-time.

As an example, the hot-swapping mechanism has been used by a K42 developer to select between three implementations of a file: 1) shared, 2) exclusive and 3) small exclusive, as described below.

Shared (Shrd): For a file accessed by multiple processes, the file's meta data and cache is most

[2]Running with and without the NULL hot-swaps is undetectable with respect to SDET throughput.

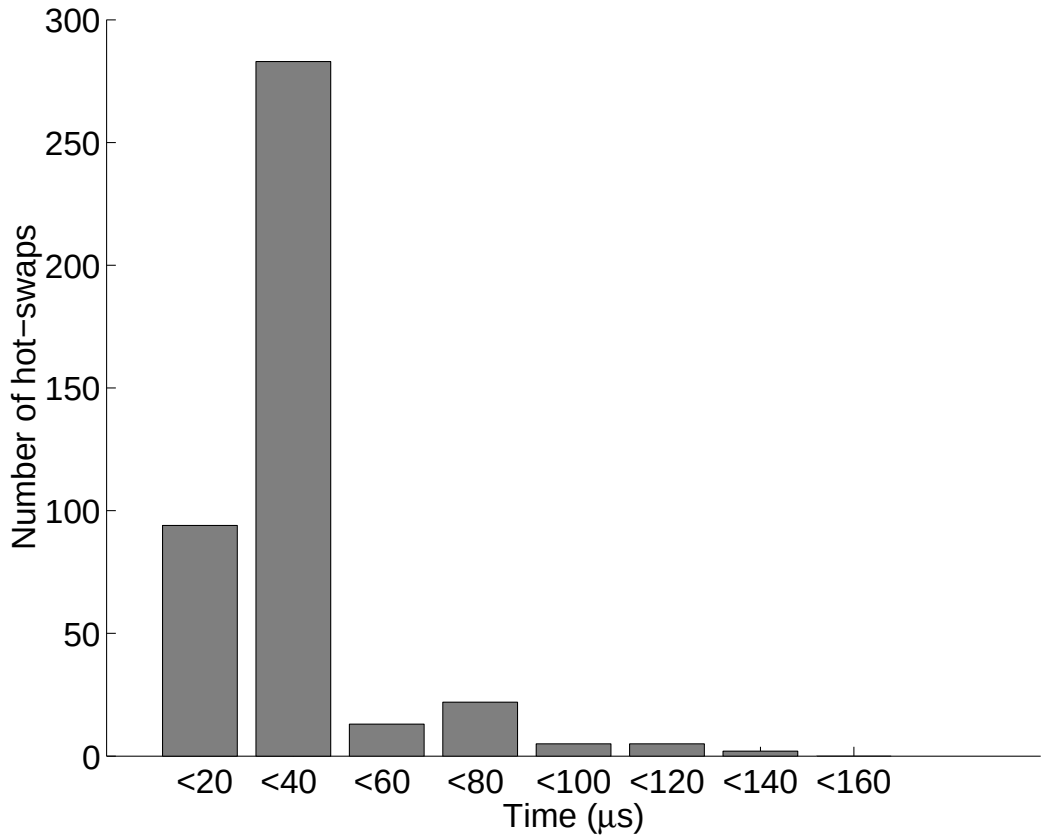


Figure 7.4: Null swap. This figure presents a histogram showing the cost of performing a null-swap of the FCM module at contended points of system execution. For each bin, there is a count of the number of swaps with completion times that fell within that bin. On average, a swap took 27.6 μs to complete, and no swap took longer than 132.6 μs.

conveniently stored in the file system server. The shared file implements this model, requiring communication with the OS on each file access.

Exclusive (Excl): When a file is accessed exclusively by one process, its file meta information can reside entirely within the application, which requires less communication with the OS.

Small Exclusive (Sm Excl): In general, file data is cached by the OS; however, access for small, exclusive files ($<$ 3 KB) can be improved by also caching the file's data within the application's address space. While this incurs a memory overhead for double caching the file (once in the application, once in the OS), this is acceptable for small files, and it leads to improved performance.

In order to take advantage of these optimizations, the developer chose to have all files, when first opened, use the most aggressive implementation and, if access dictated that this was no longer suitable, to then swap to a more appropriate implementation. Specifically, all files when opened

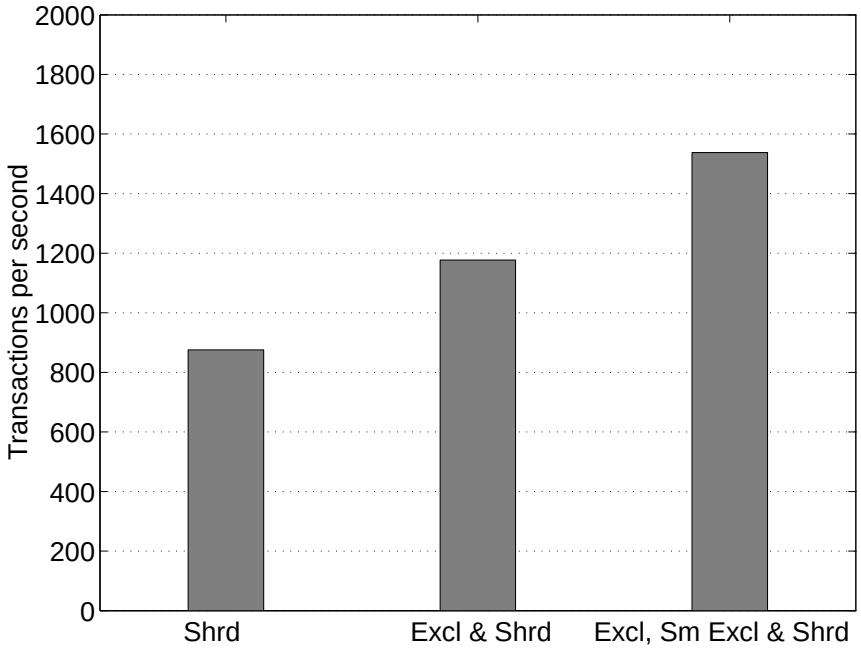


Figure 7.5: File implementation hot-swapping This figure compares the performance of Postmark under three configurations: 1) Leftmost bar (Shrd), all files use Shared Implementation, 2) Middle bar (Excl & Shrd), system uses Exclusive Implementation and utilizes hot-swapping to select the Shared Implementation when necessary, 3) Rightmost bar (Sm Excl, Excl, & Shrd), system uses Small Exclusive Implementation and uses hot-swapping to select, the Exclusive Implementation (Excl) and Shared Implementation (Shrd) as needed.

use the small exclusive implementation. If, however, the file is accessed in a manner that changes the requirements then the implementation is hot-swapped with either an instance of the exclusive implementation or the shared implementation. If the file grows beyond the small file threshold but is still accessed exclusively by one process then a hot-swap to the exclusive implementation is initiated. If concurrent access (multiple processes opening the file) to an exclusively accessed file occurs then the file's implementation is swapped to the shared implementation. Figure 7.5 shows the uniprocessor Postmark performance of three schemes: 1) shared only, 2) exclusive and shared and 3) small exclusive, exclusive and shared. In the first case the system does not utilize hot-swapping and all files utilize the shared implementation only. In the second case the system uses the exclusive implementation by default and hot-swaps to the shared implementation as necessary, resulting in a 34% performance improvement in throughput over using the shared implementation solely. In the third case, the system uses the small exclusive implementation and swaps as necessary to the exclusive and shared implementations, resulting in a 40% improvement in throughput over just using the shared implementation.

Originally, K42 implemented the above optimizations using a more traditional adaptive approach, hard-coding the decision process and both implementations into a single component. Anec-

dotally, we found that reimplementing these using hot-swapping to switch between separate implementations simplified and clarified the code, and it was less time-consuming to implement and debug.

7.6 Summary

In this chapter we have presented results which establish the effectiveness of the distributed VMM object implementations at improving the scalability of a standard system workload benchmark (SDET). We also presented the results from running the same benchmark on the Linux operating system on the same hardware platform to validate that the scalability we have achieved has not come at the expense of artificially poor uniprocessor performance. Additionally, we have shown that K42's general performance, when using the distributed implementations, is comparable to Linux on two other benchmarks, PostMark and Parallel Make.

We argue that these results show that Clustered Objects provide a means for improving the scalability of critical operating system services so as to yield measurable user-level performance improvements. The specific distributed implementations of the VMM objects presented in this dissertation are not the central result. Clustered Objects provide a framework to address measurable scalability problems through the introduction of distribution in a modular fashion to internals of a complex system.

Chapter 8

Concluding Remarks

This dissertation describes our experience and what we have learnt developing distributed implementations of services in the K42 operating system. We reviewed a series of examples of distributed implementations of objects and examined their impact using micro-benchmark experiments. We then described the basic clustered object infrastructure we developed to support distributed implementations, and discussed how the infrastructure also provides the basis to allow individual objects to be hot-swapped to new implementations. In the performance section, we established that, for general purpose benchmarks, like SDET, parallel make, and postmark, the structuring techniques used improve locality and at the same time do not prohibit competitive uni-processor performance.

8.1 Summary of Contributions

The key contributions of this work are as follows:

- We explored the application of distributed data structures in an object-oriented operating system and found a set of principles to help developers exploit distribution and locality to improve scalability. Most previous work on multi-processor performance has focused primarily on addressing concurrency with little attention to locality.

- We developed an infrastructure to simplify design and implementation of distributed objects. Such an infrastructure has proven to be helpful in coping with the associated complexity of distribution at the object level.

- We have conducted a case study, which demonstrates that a fully partitioned and localized implementation of a core OS service, virtual memory management, is possible. The existence

proof of a distributed high-performance OS service implementation is a critical result. Additionally, our work illuminates the main issues in applying distribution and provides design and code patterns to deal with them, demonstrating the synergy of an object-oriented design to the problem.

In the remainder of this chapter, we discuss each of these contributions and conclude with future work.

8.2 Principles for Developing Distributed Objects

Prior to this work, no previous object-oriented operating system project has addressed locality and distribution for SMP performance. We had gone down a number of false starts in developing distributed objects, throwing away many person-months of work. This is an anecdotal indication that such designs are non-trivial. We have found the following principles key to effective design of distributed objects:

Think Locally: Utilize a distributed component model – in our case Clustered Objects – to seek localized solutions rather than purely concurrent ones.

Be Incremental: Utilize simple techniques for developing distributed implementations from non-distributed ones, focusing on measurable performance issues, to limit scope and complexity.

Be Conservative – Encapsulate and Reuse: Be very careful in adopting complex protocols which utilize shared memory to access remote data, as innocuous code can have very subtle implications and associated errors. Where possible, encapsulate such protocols and facilitate reuse.

Inherit with Care: When using OO language support – C++ in our case – inheritance can entangle concurrency protocols across many implementations, making optimization difficult.

We discuss each of these principles in turn.

8.2.1 Think Locally

As discussed in the Chapter 2, over the last several years, the University of Toronto group has observed that a key performance characteristic of shared memory multi-processors is overheads

associated with shared data access and induced remote communications. Motivated by these observations, in this work we have chosen to pursue locality optimizations as a primary focus rather than first focusing on improving concurrency as is done traditionally. Without special support, it is easier to measure performance bottlenecks due to poor concurrency of software rather than measuring locality characteristics. However, one has a choice of how to address a given concurrency problem. The traditional approach is to improve concurrency by introducing finer grain synchronization, thus leading to reduced lock hold times. This approach, however, may not lead to better locality as it does not address the source of the problem, the underlying sharing. Another option is to treat the concurrency problem as an indicator of sharing, and seek solutions which improve locality and thus address both the lack of concurrency and, indirectly, overheads associated with sharing. Reducing sharing is a requirement for SMP software if it is to scale well independent of system size.

Key to addressing performance problems by improving locality was first developing a mental model for distributed implementations by the way of Clustered Objects. This naturally then gave us the basis on which to approach each new implementation. It gave us a concrete set abstractions with which to reason and design solutions around. For example:

- What state should be placed in the Clustered Object Representatives and hence be local?
- What state should be shared and placed in the Clustered Object Root?
- How should the Representatives be created and initialized? At what point should new Representatives participate in the global operations of the object?
- How should local operations and global operations synchronize?
- How should the CO Representatives be organized and accessed?

Based on our experiences, we have found it useful to have a model and nomenclature for distributed implementations in the developer's mind. Solutions for each performance problem could be approached within the context of the model, with the goal of providing localized hot path processing. A key utility of the model is its ability to motivate and guide the developer in achieving good locality. Our current Clustered Object model has been developed from experience and has specifically been designed to be simple. It encourages the developer to address questions like those listed above.

8.2.2 Be Incremental

It is important to acknowledge the complexity and burden in developing distributed implementations. We have found it useful to develop localized implementations as iterations on a simple non-distributed implementation through the introduction of straightforward extensions, where possible.

A typical requirement for many systems objects is to maintain some form of a mapping, where an input identifier is mapped to an output value, typically a data structure handle. Such mapping services often limit the scalability of such objects. In these cases, scalability can often be improved by introducing a local cache for the mappings while preserving the semantics and protocols of a non-distributed implementation. We have found the following to be a good first approach in improving the scalability of such objects:

1. Create a new distributed implementation by:
   (a) moving the majority of the data fields of the non-distributed version into the Root of the new implementation;
   (b) placing a cache of the critical mapping into the Representatives of the new implementation;
   (c) modifying the operations of the Representatives to access all non-mapping related data members from the Root;
   (d) modifying all operations that utilize the mapping service to first consult the local cache; and
   (e) adding appropriate code to manage and maintain the caches.
2. Measure performance using the new implementation. If the performance is not adequate, more aggressive techniques need to be considered, possibly moving additional data elements from the Root to the Representatives.

8.2.3 Be Conservative – Encapsulate and Reuse

Even when using straightforward locking strategies, asynchronous parallel software can be difficult to reason about and validate with respect to correctness. When implementing a Clustered Object for the sake of improving scalability, it can be natural to explore aggressive and subtle synchronization strategies in order to maximize performance on the critical paths. This, however, can lead to

implementations which can be very subtle and complex to reason about. Furthermore, it can also lead to implementations which are non-portable as they may rely on the memory consistency model of the architecture on which they have been developed. To this end, we found that it is best to be conservative with respect to how shared state and function are implemented across the Representatives of a Clustered Object. If possible, it is best to adopt a simple locking strategy in which a lock is placed in each Representative as well as the root and to adopt simple to use rules for acquisition. For example, all purely local operations acquire only the lock of the Representative. Manipulating global data members requires that only the root lock be acquired. For operations that must access both global and local state, the locks are accessed in a strict order (eġ; root first and then local). However, if more complex fine grain synchronization is required, then it is best to try and encapsulate these semantics in a separate component such as Dhash (see Section 4.1.4). By doing so, one restricts the pervasive fine grain synchronization from impacting the comprehension of the system object being implemented. Encapsulation allows the protocols of the encapsulated data structure to be tested, ported and verified independently. Finally, there is greater chance of amortizing the cost of development by facilitating reuse through encapsulation.

8.2.4 Inherit with Care

The final point we would like to make is based on our experience constructing distributed implementations in the context of an object-oriented language. Some object oriented features such as decomposition, encapsulation, polymorphism and specialization have proven to be very useful both in the general construction of K42 and particularly in the construction of distributed implementations. Inheritance, specifically implementation inheritance, has not proven as effective. Originally, in the Tornado project, when C++ was adopted, a decision was made to avoid using implementation inheritance. Given that there were at most three core developers and the main code base was small, this did not prove problematic. However, K42 is now a large code base developed over the years by several core programmers and many associated programmers. This has led to implementation inheritance being widely used to enable code reuse. Although this has eased the development burden and encouraged specialization, it has made it very difficult to introduce a distributed implementation for an arbitrary object. When parallel software is developed using implementation inheritance, unless foresight is given to later optimizations, synchronization semantics permeate class trees. In general, re-implementing a class which belongs to a deep class tree with a new synchronization strategy, is not possible without re-implementing all the classes

from which it inherits. This is tedious and also leads to a proliferation of classes which becomes a maintenance burden. To this end, we recommend that if implementation inheritance is going to be used, it should be done with an aim to factor out the synchronization protocols up front so that later distributed implementations can be introduced more readily.

8.3 Infrastructure for Distributed Implementations

In K42, we have made a large investment in infrastructure to support distributed and localized implementation of objects, and I was responsible for this infrastructure, which includes the Clustered Object translation mechanisms, Clustered Object base classes (which provide the default structure of a Clustered Object in terms of a Root and Representatives and the default Clustered Object Root implementations), and the garbage collection facilities. We incrementally added this infrastructure over time to move code and protocol complexity out of the implementation of individual objects and into a common infrastructure which could be re-used. As other systems start focusing on distribution and localization, the lessons learned from this infrastructure will, we believe, become increasingly relevant.

The investment made into the Clustered Object model and infrastructure has proven to be qualitatively useful and beneficial in the development and optimization of K42. At the onset of the Clustered Object work in K42, based on our initial study [3] of the object translation mechanisms of Tornado [49], we proposed an explicit model for the structure of a Clustered Object. This model, described in Chapter 5, was encoded into the basic Clustered Object infrastructure of K42 and utilized for the construction of all Clustered Objects. Below, we highlight some of the features that were found to be important and that we recommend be incorporated in any infrastructure for supporting distributed implementations. We only focus on aspects beyond those of the basic translation mechanisms which provide for the efficient and transparent mapping of a single object identifier to a local Representative.

We have found it important to establish and encode the internal structure of a Clustered Object used for general distributed implementations. We found that the notion of a "root" in a clustered object is critical for a number of reasons. First, it gives us a natural strategy to incrementally distribute an implementation, where the root is the original non-distributed implementation and we distribute just the functions that are performance critical. Second, it enables a natural implementation for what we have found to be a common case, i.e., the reps just implement a cache of

state, while there is a centralized implementation to handle all the modifications. Finally, making the root a standard part of our infrastructure has allowed us to provide standard mechanisms for maintaining information, like the list of all Representatives and a place for synchronization when destroying objects. Our experience is that it is critical to put all this complexity in the infrastructure, such as the reusable default Root implementations, rather than burden programmers of individual objects with it.

Secondly, we have found it important to design the infrastructure so that the standard model can be utilized with ease but at the same time permit knowledgeable developers to override and explore unique usages. When developing support for the Clustered Object model on top of the translation mechanisms, a layered approach was taken in which successive thin C++ classes were used to define the roles of the Root and Representatives. Each class enforced a more restricted model by providing default implementations for each role. This approach permitted the development of specialized objects, while not complicating the average developer's burden. The degrees of flexibility that has been utilized includes:

- Simple overriding of construction policies to allow the construction of "well known" static objects as well as objects which override the default lazy Representative creation in order to explicitly create one Representative per-processor at allocation time. Such objects are particularly useful when constructing base OS services created during OS initialization.

- A specialized Mediator object, described in Chapter 6, was implemented with a highly customized miss-handling behaviour in order to facilitate the hot swapping algorithm.

- A family of objects, called Arbiters, has been created which utilize a number of degrees of freedom to explore runtime call arbitration on a per-object basis.

A critical subtlety to parallel systems programming is the issue of existence locking, as discussed in Section 5.4.3. In this work we have generalized Gamsa's [48, 49] use of a semi-automatic garbage collector in order to relieved some of the burden associated with existence locking and facilitate the wider use of RCU[1] techniques. Standardizing and encapsulating the destruction protocols within the Clustered Object infrastructure alleviated the majority of developers from having to be concerned with existence locking issues. No special actions or code needs to be written when implementing new standard centralized or distributed Clustered Objects. We believe that

[1]RCU is a synchronization discipline which utilizes the event driven nature of systems software to avoid locking in the common case, see Section 5.4.3 for more details.

any multi-processor object infrastructure, especially a distributed one, can benefit greatly from a codified destruction protocol which simplify locking issues.

We have provided a facility to export the basic RCU mechanisms, implemented in the Clustered Object system and scheduling system as a general service. This has enabled the development of the hot-swapping infrastructure, a resource reclamation subsystem and a non-blocking (RCU) based hash table. We have found that by providing a simple, flexible RCU interface, *thread markers* (Section 5.4.3), we enable developers to utilize RCU based algorithms outside of the Clustered Object system. We believe that RCU mechanisms are essential in a modern system to fully exploit the event driven nature of systems software and utilize it for performance gains. Our experience of successfully constructing and using RCU mechanisms in the context of a preemptive, general purpose parallel system has served as a motivating existence proof for the Linux community which is adopting RCU support [94].

Having adopted a decomposition supporting specialization, we have extended the infrastructure to permit the development of a hot-swapping facility for replacing one implementation for another at runtime (see Chapter 6). Although our experience using this facility is limited, our infrastructure demonstrates that dynamic adaptation within the context of a high performance multi-processor operating system kernel on a per-object basis is feasible. Moreover, distributed data structures do not preclude the use of adaptation. We have shown that, with appropriate infrastructure, distributed Clustered Objects can be hot-swapped.

8.4 The Feasibility of Multi-Processor Performance

It has been widely believed that operating systems are fundamentally unable to scale to large SMP except for specialized scientific applications with restricted uses of OS functions. It has also been widely believed that any operating system that has reasonable scalability will demonstrate poor base performance and extreme complexity, making it inappropriate for generic systems and loads. These beliefs, based on current OS structures and performance, have had a large impact on how existing commercial hardware systems developed by IBM and others are constructed. In this work, we have demonstrated that a fully partitioned and localized implementation of a core OS service, virtual memory management, is possible. The existence proof of such a distributed high-performance implementation is, we believe, a major research result.

From the study of the page fault path we observe:

1. Scalability is improved through the application of techniques designed to improve locality.

2. Distributed data structures used to improve locality, by reducing the number of shared memory accesses on the critical paths, are more complex and subtle than the original centralized data structures.

Given this tension between scalability and complexity, it is important to consider how and when to make tradeoffs. Two approaches proved to be important in this regard. One key aspect was to leverage the decomposed nature of our software, specifically the object-oriented decomposition. The second was to be measurement driven.

At first one might assume that having a system path implemented over multiple objects may pose a barrier to designing and constructing an end-to-end distributed implementation. Multiple objects introduce boundaries when considering the processing of a system path, and hence lead to reasoning and implementing the path in small portions rather than in its entirety. This may seem contradictory to the motivating SMPs performance results, which illustrated that a single shared data access on a systems path can limit performance, thus implying the need to carefully construct a path in its entirety. But our experience has been to the contrary. We have found there to be a valuable synergy between object-oriented decomposition and distribution. Object-oriented decomposition helps bound the complexity of both analysis and implementation, permitting a feasible distributed implementation of a complex system service. The two main advantages have been (*i*) the opportunities afforded by the decomposition to permit an incremental approach, and (*ii*) the support for fine grain specialization.

Despite having redesigned and reimplemented multiple VMM objects to achieve good page fault scalability, the object by object approach proved to be a tractable and successful. Practically, it meant that we were able to seed the development of a completely distributed page fault path with a well-tested central implementation which established the core asynchronous protocols and requirements. This let us focus on the implications of improving performance through improved locality, as a separate concern from the fundamental design of page fault processing. Furthermore, the initial OO decomposition constrained the analysis and the way in which distribution could be introduced into the page fault path. We assert that it would have been impractical – or at the least, extremely difficult – to have initially designed and implemented the page fault paths in a distributed fashion in their entirety from the beginning. The OO decomposition leads to an iterative approach – measuring the performance for a given workload, and optimizing the first object observed to limit

performance. Optimizing one object alone was not, in general, sufficient to improve the bottom line performance, so the procedure had to be re-applied until all relevant objects were identified.

Specialization lets us isolate each unique page fault scenario, allowing us to focus only on the ones relevant to the scalability of the workloads being studied. For example, when studying SDET, the limiting page fault path was user-level page faults to long lived files such as executables and data files. As such we only needed to focus on optimizing the FCM implementation for such files. Specialization allows a developer the advantage of focusing on an optimized implementation for a constrained scenario without having to add complexity to handle the general case. This is particularly useful when applying distribution, as in general one must trade off improved performance on some operations over degraded performance on others. Thus we can leverage fine grain specialization in order to cope with conflicting demands on a shared resource to make a distributed implementation tractable.

Given the effort and complexity introduced by distributed implementations, we found it important to optimize only as performance measurements exposed a need. This ensured that we had a guiding focus for each distributed implementation, namely to improve the current performance problem. Doing so led us to adopt simple approaches that optimized the currently relevant subset of operations on the object in question. This again leads to incremental development in which complexity is bounded. The object-oriented decomposition also gave us a natural way to categorize and analyze the performance measurements gathered. Others developing performance-oriented software outside the MP domain, most notably Kernighan et al. [70] and Bentley [14], have noted the importance of an incremental measurement driven approach.

8.5 Future Work

We have identified or are currently pursuing extensions to the work of this thesis.

Locality Profiling using Hardware Support: In order to validate and ensure that the locality optimizations introduced are having the desired effect, better feedback on how the hardware is performing is required. We have started to explore how one might be able to obtain a locality profile for a performance test using hardware performance monitors available on modern processors.

Distributed Constituent Development: DHash has proven to be useful and we are pursuing its use in other Clustered Objects. We believe that the development of more constituents in

order to create a library for Clustered Object development could substantially reduce the burden of distributed object development. Part of this would include a generalized model and framework for embedding constituents.

Clustered Object Language Support: Another avenue to explore is explicit language support for the Clustered Object model. For example, the ability to use language features to explicitly define and isolate the interface between a Root and its Representative classes.

Dynamic Adaptation: In this work, we have provided a set of mechanisms for hot-swapping one object implementation with another dynamically at runtime. We have begun to explore this facility as a means for supporting general dynamic adaptation. There are many avenues and open questions to explore in this area. For example: Are there general methods for determining when a swap should occur? How can the characteristics of implementations be expressed such that a runtime system can decide when one implementation is more appropriate than another? What is necessary to use hot-swapping to apply an upgrade for a class (for example swapping all instances of a class and ensuring that new instances are created from the new definition, while preserving system integrity). One avenue to explore, with regards to evaluating when a hot-swap should be initiated, is the use of per-object runtime evaluation of performance, using hardware-supported profiling.

SMT Support: As chip-level multi-processors become more prevalent, exploring their impact on Clustered Objects, both from a basic infrastructure perspective as well as on a per-object basis will become necessary. It may prove to be beneficial to consider the reconfiguration of distributed implementations to match the performance characteristics of such machines.

Comprehensive RCU Interface: Our experience generalizing the RCU support has been very promising and has led us to pursue a new more generalized interface for exploiting quiescent states. We are pursuing a service called Quiescent Do (QDO) in which an arbitrary operation and set of processors is specified. When a subsequent quiescent state is detected on the processors, the operation is efficiently executed, on one or more of the processors depending on an invocation parameter.

Advanced RCU Support: Our current approach for detecting quiescent states across multiple processors utilizes the token passing algorithm described in this work (Section 5.4.3). We are currently exploring a new mechanism with lower latencies, which is demand-driven rather

than periodic in nature. The approach being explored is to use system constructed poll points, hardware supported synchronized clocks and timer events to ensure that quiescent states can be coordinated by time stamps across multiple processors. Such a facility would predicate a reimplementation of the garbage collection services described in this work.

OO Runtime Structure: Having observed and guided a number of new developers to K42, a general observation regarding the use of an object-oriented decomposition and infrastructure can be made: Although any given object may be tractable with respect to complexity and comprehension, we are in need of tools and support to help global comprehension, and provide the necessary context for any given object. New developers are lost in a sea of classes and inter-object protocols. Despite trying to use standard idioms, it never seems to fail that the given example that a new developer is trying to understand is a "special" case. Tools are required to help a new developer explore the runtime structure and the roles and relationships of the objects. We are currently pursuing the development of a runtime Clustered Object browser. The browser would allow a user to graphically explore the objects in existence at runtime, their relationships, performance and possible manipulate them using hot-swapping. The browser will utilize many aspects of the current Clustered Object infrastructure.

Bibliography

[1] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-16)*, pages 1–14. ACM Press, October 1997.

[3] Jonathan Appavoo. Clustered Objects: Initial design, implementation and evaluation. Master's thesis, Department of Computing Science, University of Toronto, Toronto, 1998.

[4] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Benjamin Gamsa, Greg R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and James Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[5] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the first ACM SIGSOFT workshop on Self-Healing Systems (WOSS'02)*, pages 3–8. ACM Press, 2002.

[6] Mohit Aron and Peter Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[7] Maurice J. Bach and Steven J. Buroff. Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical Journal*, 63(8):1733–1749, October 1984.

[8] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A distributed implementation of the shared data-object model. In Eugene Spafford, editor, *Proc. First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, Ft. Lauderdale FL (USA), 1989.

[9] Amnom Barak and Yoram Kornatzky. Design principles of operating systems for large scale multicomputers. Technical report, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, 1987.

[10] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[11] Amnon Barak and Richard Wheeler. MOSIX: An integrated multiprocessor UNIX. In *Proceedings of the Winter 1989 USENIX Conference: January 30–February 3, 1989, San Diego, California, USA*, pages 101–112, Berkeley, CA, USA, Winter 1989. USENIX.

[12] BBN Advanced Computers, Inc. *Overview of the Butterfly GP1000*, 1988.

[13] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In *Summer conference USENIX proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 255–275, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[14] Jon Bentley. *Programming pearls.* ACM Press, 1986.

[15] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An open environment for building parallel programming systems. In *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pages 1–9. ACM Press, 1988.

[16] David L. Black, Avadis Tevanian, Jr., David B. Golub, and Michael W. Young. Locking and reference counting in the Mach kernel. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, Software, pages II–167–II–173, Boca Raton, FL, August 1991. CRC Press.

[17] Georges Brun-Cottan and Mesaac Makpangou. Adaptable replicated objects in distributed environments. Technical Report BROADCAST TR No.100, ESPRIT Basic Research Project BROADCAST, June 1995.

[18] Ray Bryant, Hung-Yang Chang, and Bryan Rosenburg. Operating system support for parallel programming on RP3. *IBM Journal of Research and Development*, 35(5/6):617–634, September 1991.

[19] Ray Bryant, John Hawkes, and Jack Steiner. Scaling linux to the extreme: from 64 to 512 processors. In *Ottawa Linux Symposium*. Linux Symposium, 2004.

[20] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-16)*, pages 143–156. ACM Press, October 1997.

[21] Mark Campbell, Richard Barton, Jim Browning, Dennis Cervenka, Ben Curry, Tod Davis, Tracy Edmonds, Russ Holt, John Slice, Tucker Smith, and Rich Wescott. The parallelization of UNIX system V release 4.0. In *USENIX Conference Proceedings*, pages 307–324, Dallas, TX, January 1991. USENIX.

[22] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[23] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report UIUCDCS-R-89-1510,TTR89-14, Department of Computer Science, University of Illinois, Urbana, IL 61801, April 1989.

[24] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2003.

[25] Hung-Yang Chang and Bryan Rosenburg. Experience porting mach to the RP3 large-scale shared-memory multiprocessor. *Future Generation Computer Systems*, 7(2–3):259–267, April 1992.

[26] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proc. of the 1995 ACM SIGMETRICS Joint Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'95/PERFORMANCE'95)*, pages 1–13, May 1995.

[27] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, pages 12–25. ACM Press, December 1995.

[28] Eliseu M. Chaves, Jr., Prakash CH. Das, Thomas J. Leblanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, May 1993.

[29] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, pages 120–133. ACM Press, December 1993.

[30] David R. Cheriton and Kenneth J. Duda. A Caching model of operating system kernel functionality. In *Operating Systems Design and Implementation*, pages 179–193, 1994.

[31] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, 24(2):33–46, February 1991.

[32] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP-9)*, pages 129–140. ACM Press, December 1983.

[33] Andrew A. Chien and William J. Dally. Concurrent Aggregates (CA). In *Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), ACM SIGPLAN Notices*, pages 187–196, March 1990. Published as Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), ACM SIGPLAN Notices, volume 25, number 3.

[34] Christian Clémençon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on large-scale multiprocessors. In *Proceedings of the Symposium on Experience with Distributed and Multiprocessor Systems*, pages 227–246, San Diego, CA, USA, September 1993. USENIX.

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

[36] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP-5)*, pages 141–160. ACM Press, November 1975.

[37] David E. Culler and Jaswinder P. Singh. *Parallel Computer Architecture*. Pitman/MIT Press, 1989.

[38] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[39] Partha Dasgupta, Richard J. LeBlanc, and William F. Appelbe. The Clouds distributed operating system: Functional description, implementation details and related work. In *Proc. 8th Int'l. Conf. on Distr. Computing Sys.*, page 2, 1988.

[40] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds Distributed Operating System. *IEEE Computer*, 24(11):34–44, November 1991.

[41] Jeffrey M. Denham, Paula Long, and James A. Woodward. DEC OSF/1 version 3.0 symmetric multiprocessing implementation. *Digital Technical Journal of Digital Equipment Corporation*, 6(3):29–43, Summer 1994.

[42] Murthy Devarakonda and Arup Mukherjee. Issues in implementation of cache-affinity scheduling. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 345–358, Berkeley, CA, USA, January 1991. USENIX.

[43] Jan Edler, Jim Lipkis, and Edith Schonberg. Memory management in Symunix II: A design for large-scale shared memory multiprocessors. In *UNIX and Supercomputers Workshop Proceedings*, pages 151–168, Pittsburgh, PA, September 1988. USENIX.

[44] Carla Schlatter Ellis. Extensible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 106–116. ACM Press, 1983.

[45] Peter Ewens, David R. Blythe, Mark Funkenhauser, and Richard C. Holt. Tunis: A distributed multiprocessor operating system. In *Summer conference proceedings, Portland 1985:*

June 11–14, 1985, Portland, Oregon USA, pages 247–254, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[46] Steven Frank, James Rothnie, and Henry Burkhardt. The KSR1: Bridging the gap between shared memory and MPPs. In *IEEE Compcon 1993 Digest of Papers*, pages 285–294, 1993.

[47] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[48] Benjamin Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System*. PhD thesis, University of Toronto, 1999.

[49] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[50] Benjamin Gamsa, Orran Krieger, Eric Parsons, and Michael Stumm. Performance issues for multiprocessor operating systems. Unpublished, University of Toronto, 1996.

[51] Benjamin Gamsa, Orran Krieger, and Michael Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proc. 1994 ICPP*, pages 208–211, Boca Raton, FL, August 1994. CRC Press.

[52] Arun Garg. Parallel STREAMS: a multi-processor implementation. In *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, pages 163–176, Berkeley, CA, USA, 1990. USENIX.

[53] Ahmed Gheith, Bodhisattwa Mukherjee, Dilma Silva, and Karsten Schwan. KTK: Kernel support for configurable objects and invocations. Technical Report GIT-CC-94-11, Georgia Institute of Technology. College of Computing, 1994.

[54] Ahmed Gheith and Karsten Schwan. Chaosarc: kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications. *ACM Transactions on Computer Systems (TOCS)*, 11(1):33–72, 1993.

[55] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):164–189, 1983.

[56] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP-17)*, pages 154–169. ACM Press, December 1999.

[57] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 319–332, Berkeley, CA, October 2000. USENIX.

[58] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proc. 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, page 120, San Diego, California, USA, May 1991. Stanford Univ.

[59] John H. Hartman and John K. Ousterhout. Performance measurements of a multiprocessor Sprite kernel. In *Proc. Summer 1990 USENIX Conf.*, pages 279–287, Anaheim, CA (USA), June 1990. USENIX.

[60] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

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

[62] Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An object model for flexible distributed systems. In *First Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, May 1995. http://www.cs.vu.nl/~steen/globe/publications.html.

[63] Kevin Hui. Design and implementation of K42's dynamic Clustered Object switching mechanism. Master's thesis, Department of Computer Science, University of Toronto, 2001.

[64] Kevin Hui, Jonathan Appavoo, Robert W. Wisniewski, Marc Auslander, David Edelsohn, Benjamin Gamsa, Orran Krieger, Bryan Rosenburg, and Michael Stumm. Position summary: Supporting hot-swappable components for system software. In *HotOS*. IEEE Computer Society, 2001.

[65] Jack Inman. Implementing loosely coupled functions on tightly coupled engines. In *Summer conference proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 277–298, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[66] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, volume 21, pages 67–77, Portland, OR, November 1986. ACM, IEEE.

[67] David R. Kaeli, Liana L. Fong, Richard C. Booth, Kerry C. Imming, and Joseph P. Weigel. Performance analysis on a CC-NUMA prototype. *IBM Journal of Research and Development*, 41(3):205, 1997.

[68] Jeffrey Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[69] Michael H. Kelley. Multiprocessor aspects of the DG/UX kernel. In *Proceedings of the Winter 1989 USENIX Conference: January 30–February 3, 1989, San Diego, California, USA*, pages 85–99, Berkeley, CA, USA, Winter 1989. USENIX.

[70] Brian W. Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[71] Steven Kleiman, Jim Voll, Joe Eykholt, Anil Shivalingah, Dock Williams, Mark Smith, Steve Barton, and Glenn Skinner. Symmetric multprocessing in Solaris, Spring 1992. COMPCON, San Francisco.

[72] David R. Kohr, Jr., Xingbin Zhang, Daniel A. Reed, and Mustafizur Rahman. A performance study of an object-oriented, parallel operating system. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 27th Annual Hawaii International Conference on System Sciences. Volume 2 : Software Technology*, pages 76–85, Los Alamitos, CA, USA, January 1994. IEEE Computer Society Press.

[73] Orran Krieger, Michael Stumm, Ronald C. Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, volume II - Software, pages II–201–II–204, Boca Raton, FL, August 1993. CRC Press.

[74] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[75] Richard P. LaRowe, Jr. and Carla Schlatter Ellis. Page placement policies for NUMA multiprocessors. *Journal of Parallel and Distributed Computing*, 11(2):112–129, February 1991.

[76] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New York, June 1997. ACM Press.

[77] Thomas J. Leblanc, John M. Mellor-Crummey, Neal M. Gafter, Lawrence A. Crowl, and Peter C. Dibble. The Elmwood multiprocessor operating system. *Software, Practice and Experience*, 19(11):1029–1056, [11] 1989.

[78] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–80, March 1992.

[79] Roy Levin, Ellis Cohen, William Corwin, Frederick Pollack, and William Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP-5)*, pages 132–140. ACM Press, November 1975.

[80] Chu-Cheow Lim. A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions. Technical Report TR-93-063, Berkeley University of California, October 93.

[81] Susan LoVerso, Noemi Paciorek, Alan Langerman, and George Feinberg. The OSF/1 UNIX filesystem (UFS). In *USENIX Conference Proceedings*, pages 207–218, Dallas, TX, January 1991. USENIX.

[82] Heinz Lycklama. UNIX on a microprocessor — 10 years later. In *Summer conference proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 5–16, P.O. Box 7, El Cerrito 94530, CA, USA, Summer 1985. USENIX.

[83] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In Thoman L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, Los Alamitos, California, 1994.

[84] Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95*, pages 480–485, Los Alamitos, CA, USA, April 1995. IEEE Computer Society Press.

[85] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[86] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[87] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP-12)*, pages 191–201. ACM Press, December 1989.

[88] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Dept. of Comp. Sc., Columbia U., New York, NY USA, April 1991.

[89] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIGPLAN Notices*, 29(11):145–156, November 1994.

[90] Drew McCrocklin. Scaling Solaris for enterprise computing. In *CUG 1995 Spring Proceedings*, pages 172–181, Denver, CO, March 1995. Cray User Group, Inc.

[91] P. McJones and A. Hisgen. The Topaz system: Distributed multiprocessor personal computing. In *Proceedings / Workshop on Workstation Operating Systems, November 5–6, 1987,*

Cambridge, Massachusetts, pages ??–??, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1987. IEEE Computer Society Press.

[92] Paul R. McJones and Garret F. Swart. Evolving the UNIX system interface to support multi-threaded programs. Technical Report SRC-RR-21, Hewlett Packard Laboratories, September 1987.

[93] Paul E. McKenney. Read-Copy Update mutual exclusion for linux, http://lse.sourceforge.net/locking/rupdate.html.

[94] Paul E. McKenney. Kernel korner: Using rcu in the linux 2.5 kernel. *Linux Journal*, October 2003.

[95] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read Copy Update. In *Ottawa Linux Symposium*. Linux Symposium, 2001.

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

[97] Paul E. McKenney and Jack Slingwine. Read-Copy Update: Using exeuction history to solve concurrency problems. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Las Vegas, USA, 1998. IASTED/ACTA Press.

[98] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, 19(2):217–251, 2001.

[99] Bodhisattwa C. Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: experimentation with a configurable multiprocessor thread package. In *Proceedings the 2nd International Symposium on High Performance Distributed Computing*, pages 59–66, Spokane, WA, USA, 1993. IEEE.

[100] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. Medusa: An Experiment in Distributed Operating System Structure. *Communications of the ACM*, 23(2):92–104, February 1980.

[101] Eric Parsons, Benjamin Gamsa, Orran Krieger, and Michael Stumm. (De)Clustering Objects for multiprocessor system software. In *Fourth International Workshop on Object Orientation in Operating Systems 95*, pages 72–81, 1995.

[102] J. Kent Peacock. File system multithreading in System V Release 4 MP. In *USENIX Conference Proceedings*, pages 19–30, San Antonio, TX, Summer 1992. USENIX.

[103] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from multithreading System V Release 4. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 77–92. USENIX, Newport Beach, CA, March 1992.

[104] Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfelder, Kevin P. McAuliffe, Evelin A. Melton, V. Alan Norton, and Jodi Weise. The IBM research parallel processor prototype (RP3): Introduction. In *Proc. Int. Conf. on Parallel Processing*, August 1985.

[105] Robert Pike. Personal communication, 1998.

[106] David Leo Presotto. Multiprocessor streams for Plan 9. In *Proc. Summer UKUUG Conf.*, pages 11–19, London, July 1990.

[107] Richard Rashid. From RIG to accent to mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, pages 1128–1137, November 1986. Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[108] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. on Computers*, 37 8:896–908, August 1988.

[109] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proceedings of the Symposium on Microkernels and Other Kernel Architectures*, pages 73–90, San Diego, CA, USA, September 1993. USENIX.

[110] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Proceedings*

of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), pages 285–298. ACM Press, December 1995.

[111] Paul Rovner, Roy Levin, and John Wick. On Extending Modula-2 for Building Large, Integrated Systems. Technical Report 3, Digital Systems Research Center, Palo Alto, CA, January 1985.

[112] Channing H. Russell and Pamela J. Waterman. Variations on UNIX for parallel-programming computers. *Communications of the ACM*, 30(12):1048–1055, December 1987.

[113] Curt Schimmel. *Unix Systems for Modern Achitectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley Publishing Company, 1994.

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

[115] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Design rationale for Psyche, a general-purpose multiprocessor operating system. In *Proc. Intern. Conf. on Parallel Processing*, page 255, St. Charles, IL, August 1988. Penn. State Univ. Press. Also published in the Univ. of Rochester 1988-89 CS and Computer Engineering Research Review.

[116] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Evolution of an operating system for large-scale shared-memory multiprocessors. Technical Report TR 309, URCSD, March 1989.

[117] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Implementation issues for the Psyche multiprocessor operating system. *USENIX Workshop on Distributed and Multiprocessor Systems*, pages 227–236, October 1989.

[118] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. Multi-model parallel programming in Psyche. In *Proc. ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, page 70, Seattle, WA, March 1990. In ACM SIGPLAN Notices 25:3.

[119] Sequent Computer Systems. *White Paper: Sequent's NUMA-Q SMP Architecture*. http://parallel.ru/ftp/computers/sequent/NUMA_SMP_REV.pdf.

[120] Marc Shapiro, Yvon Gourbant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[121] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating Systems Concepts*. Addison-Wesley, Reading, MA, 1991.

[122] Dilma Silva, Karsten Schwan, and Greg Eisenhauer. CTK: Configurable object abstractions for multiprocessors. *Software Engineering*, 27(6):531–549, 2001.

[123] Burton Smith. The Quest for General-Purpose Parallel Computing, 1994. www.cray.com/products/systems/mta/psdocs/nsf-agenda.pdf.

[124] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma da Silva, Greg R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, B. Rosenburg, and James Xenidis. System support for online reconfiguration. In *USENIX Conference Proceedings*, pages 141–154, San Antonio, TX, June 2003.

[125] spec.org. SPEC SDM suite. http://www.spec.org/osg/sdm91/, 1996.

[126] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[127] William Stallings. *Operating Systems Internals and Design Principles Third Edition*. Prentice Hall, 1997.

[128] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[129] Jacques Talbot. Turning the AIX operating system into an MP-capable OS. In *Proc. USENIX Technical Conference*, 1995.

[130] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[131] Charles P. Thacker and Lawrence C. Stewart. Firefly: A multiprocessor workstation. *Comput. Architecture News*, 15(5):164–172, October 1987.

[132] Josep Torrellas, Anoop Gupta, and John L. Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174. Boston, Massachusetts, 1992.

[133] Linus Tovald. Posting to linux-kernel mailing list: Summary of changes form v2.5.42 to v2.5.43, http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2.

[134] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1–2):105–134, 1995.

[135] Uresh Vahalia. *UNIX internals: the new frontiers.* Prentice Hall Press, 1996.

[136] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of Globe: A wide-area distributed sytem. Technical Report IR-442, Vrige Universiteit, De Boelelann 1105, 1081 HV Amsterdam, The Netherlands, March 1997.

[137] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of 13th ACM Symposium on Operating Systems Principles (SOSP-13)*, pages 26–40. ACM Press, October 1991.

[138] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosemblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, Cambridge, Massachusetts, October 1996. ACM Press.

[139] Zvonko Vranesic, Stephen Brown, Michael Stumm, Steve Caranci, Alex Grbic, Robin Grindley, Mitch Gusat, Orran Krieger, Guy Lemieux, Kevin Loveless, Naraig Manjikian, Zeljko Zilic, T. Abdelrahman, Benjamin Gamsa, Peter Pereira, Ken Sevcik, A. Elkateeb, and Sinisa Srbljic. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, April 1995.

[140] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–80, January 1991.

[141] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson, and Frederick Pollack. HYDRA: The kernel of a multiprocessor operating system. *CACM*, 17(6):337–345, June 1974.

[142] William Wulf, Roy Levin, and Charles Pierson. Overview of the Hydra operating system development. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP-5)*, pages 122–131. ACM Press, November 1975.

[143] Chun Xia and Josep Torrellas. Improving the performance of the data memory hierarchy for multiprocessor operating systems. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.

[144] Guray Yilmaz and Nadia Erdogan. Partitioned Object Models for Distributed Abstractions. In *Proc. of 14th International Symp. on Computer and Information Sciences (ISCIS XIV)*, pages 1072–1074, Kusadasi, Turkey, 1999. IOS Press.

[145] Michael Young, Avadis Tevanian, Jr., Richard Rashid, David Golub, Jeffery L. Eppinger, Jonathan Chew, William J. Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP-11)*, pages 63–76. ACM Press, November 1987.