SCOPE: Scalable Clustered Objects with Portable Events

by

Christopher James Matthews
B.Sc., University of Victoria, 2004

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

SCOPE: Scalable Clustered Objects with Portable Events

by

Christopher James Matthews
B.Sc., University of Victoria, 2004

Supervisory Committee

Dr. Y. Coady, Supervisor (Department of Computer Science)

Dr. N. Horspool, Department Member (Department of Computer Science)

Dr. K. Wu, Department Member (Department of Computer Science)

Dr. J. Appavoo, External Examiner (IBM Research)

Supervisory Committee:

Dr. Y. Coady, Supervisor (Department of Computer Science)

Dr. N. Horspool, Department Member (Department of Computer Science)

Dr. K. Wu, Department Member (Department of Computer Science)

Dr. J. Appavoo, External Examiner (IBM Research)

ABSTRACT

Writing truly concurrent software is hard, scaling software to fully utilize hardware is one of the reasons why. One abstraction for increasing the scalability of systems software is *clustered objects*. Clustered objects is a proven method of increasing scalability.

This thesis explores a user-level abstraction based on clustered objects which increases hardware utilization without requiring any customization of the underlying system. We detail the design, implementation and testing of Scalable Clustered Objects with Portable Events or (SCOPE), a user-level system inspired by an implementation of the clustered objects model from IBM Research's K42 operating system. To aid in the portability of the new system, we introduce the idea of a *clustered object event*, which is responsible for maintaining the runtime environment of the clustered objects. We show that SCOPE can increase scalability on a simple micro benchmark, and provide most of the benefits that the kernel-level implementation provided.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Related Work

In a technological sense, concurrency can be loosely defined as simultaneous execution within a computer system. In terms of hardware this normally means more than one stream of instruction execution taking place at the same time. Concurrency presents many challenges both in terms of creating concurrency, and utilizing concurrent systems. This thesis focuses on one area of the latter; specifically, efficiently utilizing Shared Memory Multiprocessor systems. The thesis of this work is that: to ease development in the face of the system level complexities introduced by true concurrency, a user-level abstraction can be used to increase utilization without customization of the underlying operating system.

The topic of concurrency has a long history in operating systems (OSes). In order to understand the impact multiprocessors have had on operating system (OS) design and how OSes have started to utilize concurrency, it is worthwhile to begin with a brief review of early OS development. Thus, this chapter begins with Section 1.1, a brief review of concurrency in the history of OSes. In Section 1.2, we describe Symmetric Multiprocessors, a popular model of concurrent computing and some of this model's fundamental implications on system design. In Section 1.3, we overview some modern concurrency utilization techniques, such as lock free data structures, Read Copy Update, and Software Transaction Memory. Finally, in Section 1.4 we introduce an approach we believe to be key to achieving good utilization in the face of concurrent systems, namely clustered objects.

1.1 History and Context

In the 1940s up until the mid 1950s before OSes were common, users interacted directly with the computer hardware. Users manually prescribed how the resources of the machine were to be used. The cost of the labor to program systems like this was secondary to the cost of the hardware the systems ran on [32]. As prices of hardware dropped the cost of labor became more of a factor in the development of these systems; consequently, users stopped directly interacting with the hardware by developing libraries of common routines and programs such as mathematical libraries, input and output libraries, compilers and linkers [34]. The intent of these libraries was to reuse the functionality instead of rewriting it each time in the context of each application. This reuse saved labor. Carrying this desire for reuse further in these old and expensive[1] systems, and to further improve utilization of the systems the first OSes were developed by the customers of IBM.

In the early 60s, many computer vendors were developing and shipping their own OSes with their hardware platforms. Those OSes improved utilization by batching the users' work together [4, 32]. Those early OSes were controlled by a *monitor* program which was loaded and kept in the machine's memory along with other running programs. The monitor accepted a batch of user work; then serially loaded each user's program and any common libraries required. Monitors improved utilization by avoiding idleness between execution of user programs and simplified programming by providing a standard way for all users to utilize common libraries [32]. There have been identified several impacts that these early monitor based OSes had on our current model of computation both in terms of software and hardware [4, 5]:

Separation: The introduction of the monitor split the system into two. One part of the system was the monitor, the second part was the users' programs. This second part is now often referred to as *user-level*.

Privilege: To ensure the stability and security of the system, the monitor had to maintain

[1] Machines like the ENIAC cost in excess of $500,000 to create.

control of the hardware no matter what the users' program did. To solve this problem, no matter what the user-level program did, the monitor was given a higher level of privilege in the system so it could still function properly.

The following subsection describes the dawn of concurrency in those early systems and underscores its challenges in modern architectures.

1.1.1 Concurrency: Interrupts and Multiprogramming

The system monitor introduced basic user program batching. Soon after, more complex concurrency was introduced to OSes. At first, systems introduced concurrency to further increase utilization by overlapping input and output (IO) with processing. This concurrency support was in the form of a hardware based asynchronous event notification mechanism called *interrupts*. Interrupts were exploited to reduce the need for explicitly programmed IO, in which the CPU was required for the entire duration of every IO operation [32]. Using interrupts, these OSes freed the CPU to do additional processing while long IO operations were in progress. Interrupts also solved the problem of privilege by allowing the monitor to occasionally regain control of the system after an interrupt. Interrupts were able to be further leveraged to provide *batch multiprogramming*, in which more than one user-level program could be run logically in parallel.

Batch multiprogramming decreases processor idle time by enabling slow IO operations to proceed in parallel with processor execution. This meant that a fast processor would not have to sit idle and wait while a much slower IO device was servicing requests. In this model, multiple user applications were loaded in memory and started; when a given application requested IO, its execution was suspended until that IO operation was complete and the monitor switched execution to another application. When IO operations completed, the hardware signaled the monitor via an interrupt which allowed the monitor to eventually resume execution of the application which initiated the IO.

Later, this technique was extended with timers to support the interactive processing

of multiple users. Rather than waiting to switch between applications on IO events, applications were paused (*preempted*) with timers. This allowed the monitor to switch the CPU between multiple applications at a fixed time quantum which gave multiple users the perception of exclusive interactive use of the system.

Multiprogramming was a significant advance in OSes and for the first time, concurrency became an issue in the design of OSes. Despite not actually executing multiple instructions in parallel (true concurrency), multiprogramming and support for interrupts required many issues to be tackled, like: data and system consistency, synchronization of control flow, and scheduling of work.

However, "perhaps the most fundamental impact was the discovery of how complex it is to correctly implement OSes in the presence of asynchronous events and multiple executing applications." [4] In a multiprogrammed environment it is not possible to easily reason about the effects of interrelated asynchronous events. This fact is well illustrated in the comments and approaches taken by Dijkstra in the development of "THE" Multiprogramming System [14]:

> ... at least in my country the intellectual level needed for system design is in general grossly underestimated. I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous expense.

In this paper Dijkstra introduced several of strategies that the "THE" team used to deal with the challenges of concurrency. Most important of these was the notion of a *critical section*, a set of operations which require synchronization for correctness [13]. He also introduced constructs he called semaphores, which were counters that were used to guard the execution of a critical section from being executed concurrently [13, 14]. This was the first solution to the problem of achieving mutually exclusive execution of a critical section within a concurrent system.

One of the primary mechanisms used to deal with the challenges of synchronization and safety in multiprogrammed systems is hardware support for disabling interrupts. In

multiprogrammed systems the disabling of interrupts can be used to make the execution of a code path atomic. Therefore, disabling interrupts is a simple way to guard a critical section. Once interrupts had been disabled, the execution of the code path was guaranteed to proceed without preemption until interrupts were re-enabled. In these systems, interrupts were the only events that could cause current execution to be preempted.

However, with the advent of modern multiprocessors, disabling interrupts on a single processor was no longer sufficient to ensure atomicity. A multiprocessor has multiple CPUs which can concurrently and independently execute instructions. Rather than simply interleaving instructions in response to interrupts, multiple applications and system requests can be executing in a *truly concurrent* fashion.

Hence, matters were further complicated with the advent of true concurrency: where systems have more than one CPU and can execute instructions completely in parallel. The next section describes Shared Memory Multiprocessors (SMMPs), and Symmetric Multiprocessors (SMPs), a common configuration for systems with more than one processor and the platform of interest for this thesis. We also look at sharing which is a phenomenon that naturally results from caching on these systems.

1.2 Shared Memory Multiprocessors

One model of hardware concurrency that has recently become exceedingly popular is Symmetric MultiProcessors (SMP)[2], a model in which more than one general purpose processor executes instructions. SMP is a type of Shared Memory Multi Processor (SMMP), where all of the processors are general purpose and can operate in one shared memory space. This means that every processor can access the same memory as every other processor. SMP machines offer a programming model which is a natural extension of a typical uniprocessor; a single shared address space. This has resulted in most general purpose multiproces-

[2] As opposed to Asymmetric Multiprocessors which have special purpose processors executing processor specific tasks, for example the graphics processors (GPU) on a video card.

SMP machines		
System	Processor	Cache Information
Intel Server	2 processors	16 kilobytes non-shared L1 cache, 1024 kilobytes non-shared L2 cache
AMD workstation	2 processors	64 kilobytes non-shared L1 cache, 256 kilobytes shared L2 cache
AMD workstation	1 dual-core	64 kilobytes non-shared L1 cache, 512 kilobytes non-shared L2 cache

Table 1.1. *Three common SMP machines used in the year 2006*

sors today being SMPs. Table 1.1 shows some sample SMP machines that are commonly available today.

Given the familiar programming model, it was natural to develop OSes for SMPs as an incremental extension to uniprocessor OSes. Although this approach was the natural course of development, research has shown that without some extra structure it is not necessarily the best approach with respect to yielding high performance SMP OSes [2, 4, 6, 11, 19, 25, 28].

Until recently, SMP systems were more expensive than uniprocessors systems so they were mostly reserved for server systems and scientific computing applications; however, the cost of multiprocessor systems has steadily declined. Furthermore most of the major chip manufacturers are using dual-core technology, which places more than one processor in a single chip [1]. Chip manufacturers have found that instead of increasing the speed of chip, it is easier to provide more processors to increase throughput. As SMP systems become more and more common, the corresponding demand for effective SMP programs will increase. Unfortunately supplying support for highly effective concurrent programs is hard.

As we will show in the following sections, SMP proves to be complex for system designers to achieve good *scalability*. Mathematical formula aside, the definition of the term *scalability* used in this work is: the desirable property of a system, a network or a process, which indicates its ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged [8]. In the context of demands for concurrency, we take

this definition to mean maintaining good hardware utilization and throughput as the load increases.

The rest of this section introduces caching, a hardware mechanism used to speed up memory access on modern processors. Sections 1.2.2–1.2.4 describe sharing, a troublesome phenomenon associated with caching on SMP systems. Finally we describe state of the art support for true concurrency in modern systems.

1.2.1 Caching

Generally, processor technology has been faster than the memory it accesses [32]. In systems with large disparities in memory access speeds, caching is often used to mitigate some of the performance effects. This is normally attributed to cost: the faster the memory has to be, the more it costs – thus large memory systems use relatively slow memory. This plays out throughout the entire memory hierarchy. The memory of most relevance to this work is the cache memory that sits between the processor and main memory. Like their uniprocessor ancestors, modern SMP systems use caches to increase their memory access performance. For example Figure 1.1 shows a simple two processor system which takes the form of a *level 1* (L1) cache (the fastest, but smallest) on each processor, then (a slower, but larger) *level 2* (L2) cache that is shared by all the processors. Each one of these caches must have the correct data in them when accessed, so synchronization for cache coherency must take place between L1, L2, and main memory.

When some data in memory is needed by one of the two processors, each cache is checked in order L1, then L2. If the data is not in the caches, it is taken from the main memory, and added to each cache of the requesting processor, evicting something older back to the main memory. The next time that data is needed it will already be in the cache (if it has not been evicted by some other memory request).

This description is a simplification of what the hardware actually does. In reality caching is much more complex. To provide better performance, strategies like prefetching and customized eviction policies can be applied. Strategies can vary between architectures,

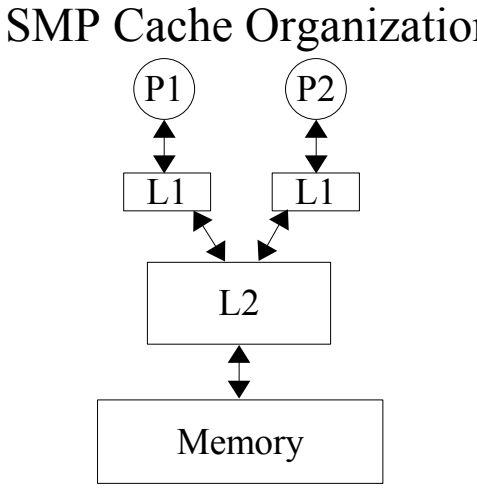

Figure 1.1. *One possible abstract representation of cache and memory layout for a SMP machine. Processors P1 and P2 independently access the non-shared L1 caches, which are synchronized with a shared L2 cache which is further synchronized with memory.*

and even between manufacturers. All this, combined with the complexities of synchronization for coherency between caches and between cache and memory makes it extremely hard to predict the behavior of a system with respect to cache behavior.

The atomic unit used to access data in caches is called the *cache line*. Typically 64 bytes to 512 bytes long, a cache line is filled with contiguous data from the memory its cache is fed from. Cache lines are the lowest level of granularity the cache is managed at. Therefore, cache lines are the units by which evictions take place. When a particular memory address is requested, a minimum of a full cache line with that address' data in it is loaded into the cache.

1.2.2 Sharing

When two or more processors access the same piece of data, we call that *sharing*. With regard to caches, sharing can carry a subtle but highly significant penalty. When data is shared, it has to be synchronized between all the other processors in the system. For example, in a four processor system with processors A, B, C and D: if A writes to memory

address x, then a message must be sent to B, C and D telling them that their copy of the cache line associated with address x is no longer valid; however, now the cache copies of data associated with x in B, C and D are invalid, so when they go to access x, they each have to (re)get data associated with x from main memory. Every time a write happens to x, this process has to happen. Through Read-only data does not incur this penalty, sharing introduces serious overheads to write operations, especially if they are frequent and on different processors. Unfortunately, sharing is inherent in many common data structures, and more generally in many common workloads. Thus sharing at the level of cache lines is a significant obstacle to effective, high quality, truly concurrent software.

1.2.3 False Sharing

As mentioned above, the granularity of operations on a cache is of the size of a cache line. This granularity leads to a second unfortunate phenomenon. When one piece of data is shared all other data in the cache line are implicitly shared. This is known as *false sharing*. Unfortunately, false sharing is costly because even when data is not being intentionally shared, some neighboring data may cause the same sharing effects as real sharing would. Unless intentionally and explicitly organized, the placement of data on cache lines is not normally known a-priori, so the effects of false sharing could strike anywhere. The next section describes *locality*, and its relationship with sharing.

1.2.4 Locality and Sharing

Achieving scalable performance requires minimizing all forms of sharing. Looking at this in another way, minimizing sharing can be thought of as maximizing *locality*. On SMPs, locality typically refers to the degree to which locks are used or contended and data (including the locks themselves) are shared amongst different processors. The less the shared data, the higher the locality associated with a processor's access patterns. Maximizing lo-

cality on SMPs is critical, because even small amounts of sharing or false sharing can have a profound negative impact on performance and scalability [11, 19].

Figure 1.2 shows an example of how sharing takes its toll on the standard benchmark SDET [17]. The Linux version suffers from sharing, so as more processors are added, per processor utilization eventually lessens and then the toll of sharing ultimately becomes so great that throughput actually drops. The other line in Figure 1.2 is an implementation in K42 that was designed to control sharing by using per processor data, it performs much better as the number of processors increase. As processors are added on those systems, throughput still rises almost linearly. With these kinds of scalability characteristics, reducing sharing is necessary for writing highly concurrent systems.

Previous work has shown that with considerable effort, one can reduce sharing in the OS in the common case [4]. This allows for 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 within the control of the OS designer is the data structures and algorithms employed by the OS. However, [19] observes that, prior to addressing specific data structures and algorithms of the OS in the small, a more fundamental restructuring of the OS can reduce the impact of sharing by minimizing sharing in the OS structure." [5]

That work uses an object-oriented model to create independent instances for OS resources. Their model has a desirable concurrent property: accesses to independent parts of the system are serviced by independent data structures. Inherently, this approach helps to control sharing by possibly reducing the amount of shared data encountered during the fulfillment of a request.

"However, the above approach does not completely 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 [6], a multiprocessor OS constructed using this design. Assume a workload that simulates multiple users issuing

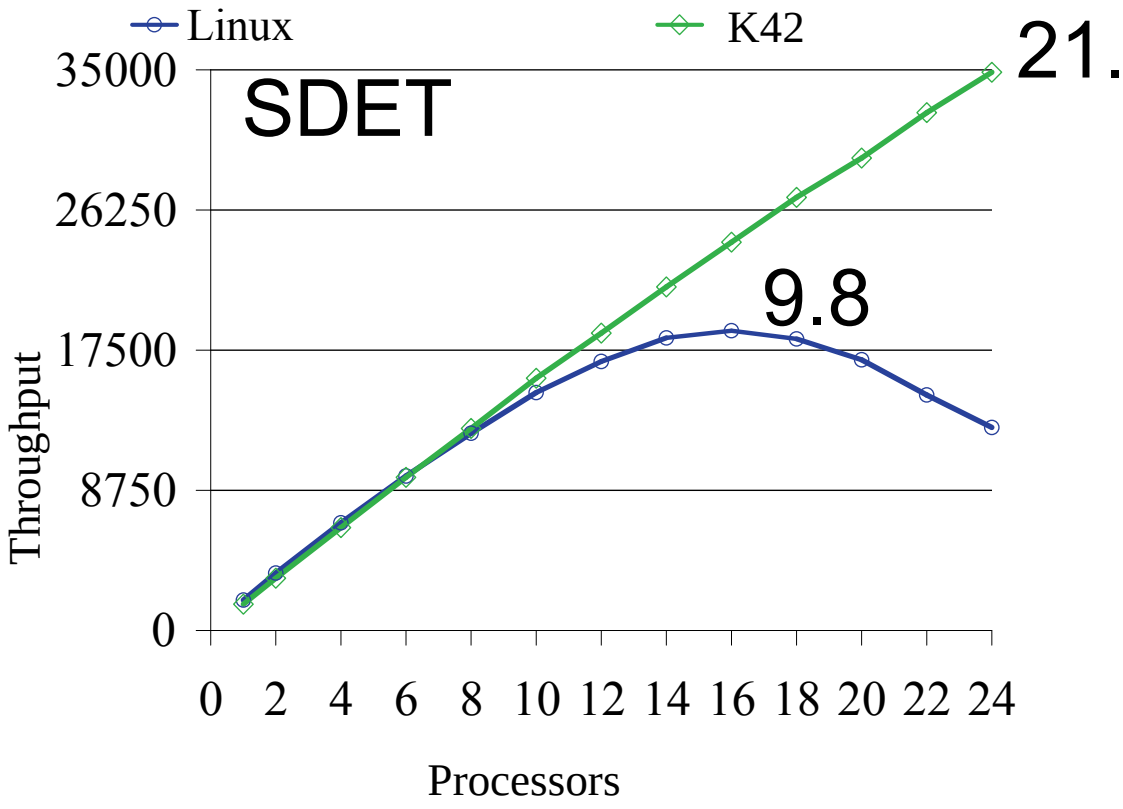

Figure 1.2. *Throughput of a standard benchmark on two different OSes taken by the K42 team [5]. The effects of sharing limit scalability on Linux, K42 is not affected.*

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 deliberately 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 [as depicted in Figure 1.2.]... Ideally, the throughput should increase linearly with the number of processors." [4]

They show that on closer inspection the workload induces sharing on OS resources, thus limiting scalability. The K42 group contends that in order to ensure the remaining sharing does not limit their OS's performance, distribution, partitioning and replication must be used to remove sharing in the common code paths [4]. Using distributed implementations for key virtual memory objects, and running the same workload as above, they found that the OS yields a 3.9 times throughput at four processors and a 21.1 times throughput at 24 processors [4].

Thus, it is possible with considerable effort, to reduce sharing and produce scalable systems. Over time, mechanisms to help deal with concurrency effectively have evolved to ensure both scalability and safety. The next section outlines some of these mechanisms.

1.3 Modern Solutions for Utilizing Concurrency

Conceptually, the notion of a lock which guards a critical section is simple; however, there are more complex ways to provide concurrency without relying on the mutual exclusion of critical sections. However, these methods each have their own associated advantages and disadvantages. The following subsections overview this theme of ways to increase utilization. Section 1.3.1 starts with locking, and Section 1.3.2 describes the more complex approach of lock-free data structures. Then we discuss more recent methods like Read Copy Update in Section 1.3.3 and Software Transactional Memory in Section 1.3.4.

1.3.1 Locking

Most modern OSes have settled on the semantics of a lock for synchronization. The fundamental operations on a lock are *acquire* and *release*. Each critical section is associated with a lock. At the start of the critical section an acquire is performed on the associated lock, then after the critical section, a release of the same lock is performed. Implementation of the lock must ensure serial execution of the critical section. When the release operation is executed, another process is allowed to enter the critical section. As lock implementations advanced, they took into account notions of *forward progress*, attempting to provide guarantees about how the processes attempting to execute a critical section would progress. For example, all processes will eventually execute the critical section, or processes will execute the critical section in FIFO order. Later solutions also attempted to account for the performance of the primitives, ensuring efficient execution on typical hardware platforms. The exact semantics as to which process may enter is implementation dependent and has direct bearing on the forward progress properties associated with the implementation.

In order to try and increase concurrency, variations on the lock semantics have been introduced. These include, reader-writer locks where two independent types of acquisition are introduced: read and write [24]. In order to improve concurrency, readers are allowed to operate concurrently on the data structure as long as they do not modify it; but, a writer must execute mutually exclusively with respect to all other readers and writers.

The implementation of locks to achieve mutual exclusion on general purpose SMPs typically synchronizes processors via shared variables. A great deal of effort was spent in studying the performance of SMP locking techniques [27]. These efforts concluded that standard locking exhibits poor locality, and that special locking techniques need to be used that are SMP aware [24, 27].

The mutual exclusion provided by locks is a simple way to provide safety, but by blocking access to data you limit the scalability of a system fundamentally. With some effort more elaborate methods can actually remove the need for locking altogether and possibly provide a more scalable system. The following section highlights one of these approaches.

1.3.2 Lock-Free Data Structures

The scalability of a data structure can be limited if mutual exclusion is used. A lock-free data structure is one that allows concurrent access without using mutual exclusion beyond simple atomic operations [7]. Before a general methodology for creating lock-free data structures was created, specific data structures like queues, stacks, linked lists, union-find sets, and for algorithms like set manipulation and list compression were found to have lock-free implementations [7]. This work was unified by [22] who showed that there were universal primitives that all of the above used. Basic lock free algorithms work by copying a data structure, making changes to that private copy, then updating the public pointer to the data structure to the address of the private version [7]. If the pointer has already been changed, then the update must be restarted. This method has many problems: that of having to make a copy of the entire data structure each time it is updated. If there is more than one pointer to the data structure, they all have to be tracked and updated, and there is no guarantee of forward progress.

In [7], they use the cooperative technique to organize threads; so, if one thread is writing to a location, and another wishes to write to that same location, the second thread helps the first thread finish, then executes its own write. This allows them to make the same forward progress guarantees that locks can. [7] also introduces what they call a caching method which only copies parts of the data structure.

In regards to performance, lock-free data structures have excellent read performance because there is no overhead associated with reading data. However, write performance is much worse than normal, there is a significant extra amount of work that must take place to organize the writes, and the overhead gets heavier as more writing happens; furthermore, this method does not take into account sharing. The next section introduces Read Copy Update, a lock free method that aims to perform better in write intensive situations.

1.3.3 Read, Copy, Update

Read Copy Update (RCU) was designed to remove some of the drawbacks of lock-free data structures and other update methodologies. RCU does this by relaxing the requirement that data be written as soon as the write is triggered [26]. Instead, RCU waits until a time when the write can be done without interfering with anything that depends on that data.

RCU introduces the idea of a *quiescent state*, a point in the thread where it no longer makes any assumptions about any guarded data structures [26]. They also introduce the idea of a *quiescent period*, a period of time in the program in which every thread passes through at least one quiescent state [26].

RCU tracks these quiescent states throughout the system. Then, RCU triggers writes in a batch on each thread as it enters its quiescent state. As each thread passes through a quiescent state, it no longer makes any assumptions about the old data, so by the end of a quiescent period all assumptions made about the old data are gone and the system is effectively working on the new data.

One interesting property of RCU is that because it batches its writes, the more writes that happen the less the per write overhead [26]. Effectively, this means it may work better than the lock-free data structures mentioned above under heavy write loads. In regards to performance, RCU is comparable to modern locking mechanisms [26]. When implementing RCU the designers also took into account sharing, so the RCU mechanisms do not cause much sharing. RCU does not however help the client code reduce sharing. RCU requires integration with the system it is running on to detect the quiescent states.

But, sometimes programs cannot tolerate stale data, or cannot drastically alter the system they are running on, in these cases RCU is not applicable. Another update methodology is Software Transactional Memory which tackles the problem from a slightly different angle.

1.3.4 Software Transactional Memory

Software Transaction Memory (STM) provides transaction-like semantics for critical sections. Conceptually, everything in a critical section happens as a single atomic operation that either succeeds (*commits*) or fails [31]. In the event of a failure, a retry can be issued.

STM is based on a design for Hardware Transactional Memory [31]. Initially the first STM could only work statically, and had unusual hardware instruction requirements; however, more recent implementations have fixed those problems [20]. STM systems also have a trade off between providing either fast single access or fast batch access [20].

In regards to implementation, STMs are similar to how lock-free data structures work. STMs use a modified cooperative technique like lock-free data structures; however, instead of working at the data structure level, STM defines a general list of memory cells, which can be written to, or read from [31].

In regards to performance, STM has been shown to be competitive with locking [20]. But it does not address the issue of sharing. None of the solutions for utilizing concurrency outlined so far focus on increasing locality. The next section introduces clustered objects, a mechanism to help control sharing and promote locality.

1.4 Clustered Objects: a Proven Solution to Scalable OSes

"Despite decades of research and development into SMP OSes, achieving good scalability for general purpose workloads across a wide range of processors, has remained elusive. Sharing lies at the heart of the problem." [4] The rest of this section describes the sharing problem and motivates clustered objects, a proven system used to maximize locality. Section 1.4.1 introduces clustered objects and Section 1.4.2 discusses where locality management is most useful, and then introduces Scalable Clustered Objects with Portable Events, our solution to user-level locality management.

Sharing introduces barriers to scalability. In OSes, sharing comes from three main sources [4].

1. Sharing can be implicit in the workload. Programs use shard variables, and access shared resources in the system. But beyond simple shared variables,
2. sharing can arise from the data structures and algorithms used in a system.
3. Sharing can occur in the physical design and protocols utilized by the hardware. For example, some systems utilizing a single shared memory bus can cause sharing.

In general one of the goals of OSes is to provide a framework for efficiently utilizing computer hardware. To achieve this goal of facilitating hardware utilization, the OS provides an abstract model of a computer system and an interface with which the program can access this model. A critical issue in the development of OSes is to enable efficient application utilization of hardware resources [4]. To ensure that a parallel application at user-level can realize its potential performance, all services in the system domain that the concurrent application depends on must be provided in an equally parallel fashion [4, 33]. More simply, the scalability of an application can be limited by the underlying OS.

Though the work on clustered objects in OS's domain has proven effective [4], the problem of sharing does not just exist at the OS level. Parallel applications that want to utilize SMP systems to provide real performance face many of the same challenges that an OS faces.

1.4.1 The Need for Speed: Everyone Has It

As we have seen, the development of high performance, parallel systems software is not trivial. The concurrency and locality management needed for good performance can add considerable complexity to any system. The fine grain locking used 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 these data structures in ad hoc manners. In this work, the term *per processor data structures* refers to the use of a separate instance of a data structure for each processor. Clustered objects were developed as a model

of partitioned objects to simplify the task of designing high performance SMP systems software [28].

A key to achieving scalability and performance on a multiprocessor is to use per processor data structures whenever possible, so as to minimize inter-processor coordination and shared memory access [28]. 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 intended to improve performance by enabling distribution, replication and partitioning of stored data. In general, access to 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 the *scatter-gather* operations that distribute the data, and aggregate previously distributed data.

As explained in more detail in the next chapter, in a partitioned object model, objects are composed of a set of distributed *representative objects* [4, 30]. Representatives are spread throughout the system, when a request is made it is redirected to a representative that is best suited to accept the request. As an aggregate, the representatives produce the system wide functionality of the partitioned object.

The partitioned nature of clustered objects makes them ideally suited for the design of scalable shared memory multiprocessor system software. “This type of software 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 [18] and proxy [29]; however, they have been carefully constructed to avoid any shared front end, and are primarily used for achieving data distribution.” [4]

Following previous work: “use of the word *distributed* throughout refers to the division of data across a shared memory multiprocessor system. In this context, distribution does not require message passing, but rather, distribution across multiple memory locations all

of which can be accessed via hardware supported shared memory." [4] Data is distributed across multiple memory locations in order to change cache line accesses patterns to control sharing which may ultimately increase concurrency.

This thesis extends clustered objects from the OS kernels, where it currently is hosted, to user-level which is a more easily accessible environment for the average programmer.

1.4.2 SCOPE

Clustered objects present a unique mechanism for systematically enhancing processor utilization. As we have seen in Section 1.3.2, mechanisms like lock free data structures and STM are user-level constructs, and thus can be used in most programs; however, unlike these mechanisms, to this point, all implementations of clustered objects have been realized in OS kernels or have relied heavily on non standard kernel support.

Clustered objects have never been implemented in a system independent manner at user-level. So, is it possible that: **in the face of the system level complexities introduced by true concurrency, a user-level abstraction can be shown to increase utilization without control of the underlying system.**

In this context, we take hardware utilization to mean the increased throughput caused by CPUs not having to wait during memory latency periods. In this context, the underlying system is taken to be an OS, and control of the OS is taken to mean the ability to change the OS, instead of just utilizing the services it provides.

We will validate this thesis by re-implementing a kernel-level implementation of clustered objects as a user-level library we call SCOPE. Then we will check to see if the same fundamental benefits of clustered objects still apply to the user-level implementation.

1.5 Summary

The evolution of OSes from simple monitor systems with batch multiprogramming to modern day OSes with SMP support has required drastic change with respect to how concur-

rency is dealt with in order to provide better resource utilization. Through the years, the solutions to problems faced during the evolution of OSes contributed to modern concurrent programming models. Although mutual exclusion based locking schemes are simple and the most common form of concurrency control; lock free data structures, STM, and RCU provide interesting alternatives to mutual exclusion based system, and hold promise to handle heavy concurrency better by improving utilization and batching write workloads.

From a low level perspective, the caches used in modern multiprocessors complicate concurrency further. Sharing and false sharing can cause data access to effectively be much slower. A highly aware programmer is able to avoid these problems through the thoughtful application of fine grained mechanisms, but without the added structure provided by models like clustered objects, these solutions are at best ad hoc, and instance specific, and provide little reuseability, or evolveability.

Clustered objects appear to offer improved scalability with respect to the problems faced by concurrent systems. Chapter 2 now takes a closer look at the clustered object model, provides implementation details of their concrete manifestation in an OS, and discusses the expected benefits of clustered objects in general.

Chapter 2

Background: The Clustered Object Model

This chapter provides an overview of the clustered objects model and its basic operation within K42's kernel clustered object system. We begin with an explanation of partitioned objects and de-clustering, the key ideas behind the clustered objects model. In Section 2.1 we describe the basics of the clustered object model. In Section 2.2 we explain how clustered objects have been implemented, and specifically how they can be accessed. Then, in Section 2.3 we overview the benefits that the clustered objects model provides.

2.1 Object Models

This section provides an overview of the clustered object model on a conceptual level. We start with an overview of *partitioned objects*, a model in which objects are broken up into parts. We then explain how clustered objects are actually a model of partitioned objects. Then, finally, in Section 2.1.3 we highlight the clustered objects model and define some of its basic entities.

2.1.1 Partitioned Objects

The term *partitioned objects* refers to a strategy commonly used in systems to break up an object into local components to aid distribution. In the context of distributed systems

this strategy is also known as the proxy pattern [29, 30, 36]. In the context of systems software this strategy of distributing data to achieve better performance has been called *de-clustering* [28]. De-clustering has been shown to increase locality in many situations [19, 28] including in scheduling [2], memory allocation [25], and synchronization [27].

To give a high level overview of the impact of de-clustering with respect to scalability, imagine two different systems trying to satisfy a large number of requests. System (a) has lots shared data elements on the path that satisfies the requests. In this case each request must wait to control the shared data used in its path before proceding. System (a) experiences poor scalability as the number of concurrent requests increases; however, system (b) could reduce the common data between requests, which reduces sharing, and therefore system (b) experiences better scalability.

In a partitioned object model the implementation of an object is broken into smaller logical units that are closer to the caller, each of which is able to act on behalf of the whole object. As depicted in Figure 2.1, although externally the client sees an object with a single interface, internally the object is made up of several different elements. When a request is made upon the object from its external interface, some mechanism redirects the request to the appropriate internal element where the request is then satisfied.

In the SOS distributed OS, partitioned objects are implemented as what was called *fragmented objects* [30]. A fragmented object is a system wide object that has a local fragment on each node in the distributed system. When a node wants to make a request on the system wide object, it does so by accessing the local fragment, which either has all the necessary logic to satisfy the request locally, or will send the request somewhere else in the system to be satisfied.

The clustered object model utilizes the partitioned object model to de-cluster objects [4, 19, 28]. This presents very different challenges in SMP architectures than for their distributed systems counterpart as issues such as communications overheads, and failure modes are very different [28].

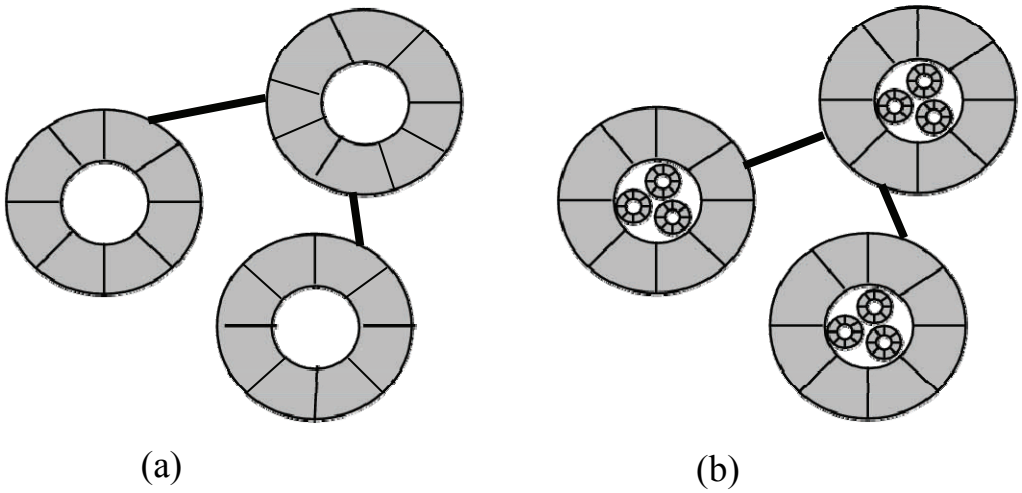

Figure 2.1. *(a) represents regular objects. The shaded outer interface (grey ring) provides a barrier to the data on the inside. In (b), a partitioned object maintains the same interface, but inside is made up of several more objects instead of data.*

2.1.2 Clustered Objects as a model of Partitioned Objects

The goal of clustered objects is to take the generally ad hoc manner in which de-clustering has been applied in previous systems and facilitate a more ubiquitous approach. To that end, clustered objects is a model of partitioned objects that helps the client apply de-clustering to data. A clustered object presents the illusion of a single object to the client, but is actuality composed of several component objects. Each component handles calls from a specific subset of the machine's processors. Inside every clustered object, the notion of global information and distributed information is made explicit. Each type of data is separated out into different classes, of which the distributed data classes may have per processor instances. When a request is made, logic in the clustered object system allows the programmer to decide where the request will be directed, and how to ultimately satisfy the request. What data is global, what data is distributed, and how the distribution and aggregation of the data occurs is defined by the creator of a clustered object and is transparent to the client. This customization is what helps make clustered objects easier for programmers to build scalable objects and therefore services that will scale better.

2.1.3 The Clustered Object Model: Roots and Representatives

Clustered objects are referenced by a common clustered object reference that logically refers to the whole clustered object; however, each access to this common reference is automatically directed to a local *representative* (rep) [19]. Figure 2.2 shows a simple case with three processors marked P1, P2 and P3. Each processor accesses the clustered object through a global reference, then the clustered object system redirects the call to a local representative assigned to that processor.

Every clustered object is made up of a *root* and one or more representatives. These components correspond directly with global and distributed data. Roots contain global data, reps contain instances of distributed data and the methods that control and aggregate the distributed data. A root is not directly accessible except through its representatives; so, in this respect representatives are responsible for providing local access to global data. Roots themselves are responsible for dictating which reps are assigned to handle requests in any given *locality domain*. In this context we define a locality domain as the memory used by a particular processor. The class diagram in Figure 2.3 shows the basic classes that comprise a clustered object. Each root has one or more reps to satisfy incoming requests.

2.2 Clustered Object Implementation

This section overviews some implementation details of Clustered Objects, and then details of the K42 implementation of clustered objects. In Section 2.2.1 we describe the Clustered Object Manager, the object responsible for coordinating the clustered object runtime in K42. Sections 2.2.2 through 2.2.4 review the mechanisms typically used for lookups, identification and accesses of clustered objects: translation tables, clustered object IDs, and the dereferencing system.

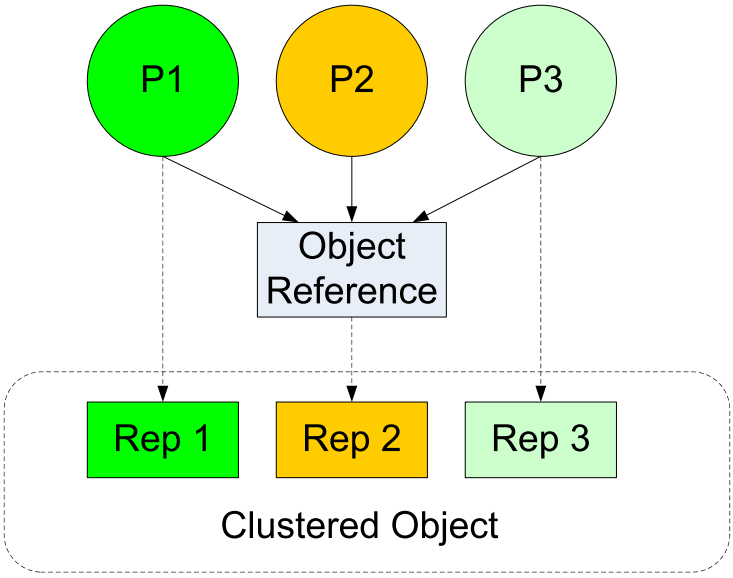

Figure 2.2. *Processors P1,P2 and P3 access a clustered object through its common reference. The request made to the global reference redirects the call to a different rep for each processor, so that each processor is using a different rep. The root is not shown in this figure.*

2.2.1 The Clustered Object Manager

In K42, the *Clustered Object Manager* (COSMgr) is responsible for coordinating and controlling the Clustered Objects runtime environment and the clustered objects life cycle [4]. The COSMgr's responsibilities include:

- system initialization including all of the object tables,
- clustered object allocation,
- clustered object deallocation (via garbage collection).

To ensure scalability of the Clustered Objects facility, the COSMgr is itself a clustered object and hence uses many of the services that it provides. As might be expected, this leads to a complex and very incremental development and creation process. Because the COSMgr is a clustered object, it must abide by the same rules as all other clustered objects, and experiences all of the same benefits of being a clustered object mentioned in Section 2.3.

2.2.2 Translation Tables

Two of the key elements of the COSMgr are the *local translation tables* and the *global translation table*. These tables store the basic information needed to access a clustered object. Both tables hold translation entries, a small set of data that is needed to access and manage a clustered object in a performance conscious manner. A global translation entry is three machine pointers long, whereas a local translation entry is only two pointers long. The Global Translation table is a single array that contains enough elements to have one entry for each clustered object in the system. The local translation tables each have the same number of elements as the global table; however there is a local table for each locality domain in the system. The local table's corresponding entries contain the data necessary to access the representative of the clustered object that is assigned to the locality domain from which the current request originated.

2.2.3 A Clustered Object's ID

In the current implementation of clustered objects in K42, a clustered object's system wide unique ID is its index into these translation arrays. For example, a clustered Object with ID of 5 would have an entry in the global array at element 5, then in each locality domain the local translation table for that domain will have the corresponding representative's entry in element 5.

One problem with this system is that elements cannot be reused in the arrays, or else the unique identifier might be used more than once. As new clustered object runtime systems are implemented there is a drive to separate clustered object's IDs from their associated lookup mechanism [5].

2.2.4 Accessing a Clustered Object

One of the important features of K42's Clustered Objects facility is their optional *lazy* initialization. When there is a large number of processors on the system, having unused

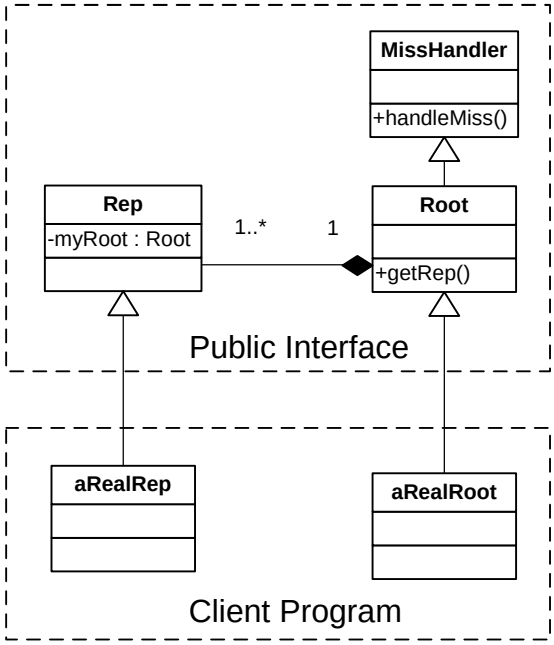

Figure 2.3. *The basic clustered object classes. Each root has one or more reps.*

representatives for each processor can drastically and unnecessarily increase the memory usage of a clustered object and decrease scalability. To solve this problem, when a clustered object is created, only the root is allocated, then the first time a clustered object is accessed on each processor that processor's representative is created. When a local representative is not active in the local table, we call that *suffering a miss*. When a miss happens, the global table is consulted for a *miss handler*. Miss handling functionality is built into a base class of the clustered object roots, so the global table returns the root for the particular clustered object that suffered the miss. Then it is the miss handler and root's responsibility to take some action.

The particular course of action the miss handler takes is dependant on the desired outcome. In the common case of a lazy initialization, a rep would be created, the local table entry for the processor that is suffering the miss updated, and the rep is returned to the system to begin execution on the desired method. If a certain degree of clustering was required, for example, one rep per 4 processors, the miss handler could assign local table

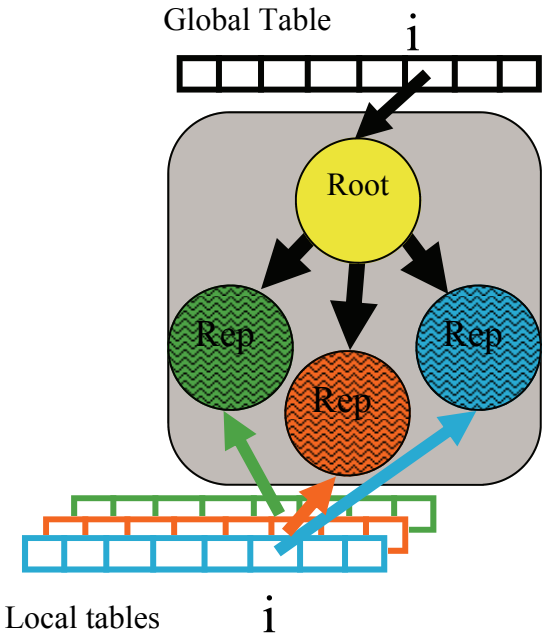

Figure 2.4. *Processors P1,P2 and P3 accessing a clustered object through its common reference (i). The first time they try to access it, each suffer a miss. The global table on the top is consulted for the miss handler which assigns one of the reps on the bottom, and changes that processor's local lookup table on the bottom to match that rep. From that point on, the local table's value is used to perform a quick lookup.*

entries and create reps accordingly. The miss handler can even not change the local table entry, causing future access of the clustered object on that processor to continue to miss. In Tornado (K42's predecessor), the miss handling process takes approximately 150 (MIPS) instructions [19], so some overhead is incurred to handle a miss, though not enough to preclude the use of this mechanism for general purpose dynamic actions [19].

The implementation of the mechanism explained above is not trivial to achieve while maintaining language type safety. When a program using clustered objects is compiled, we want the compiler to treat the methods in the clustered objects as those that are called when a clustered object is dereferenced; but in actuality, what we want to happen is for either the method to be called or the miss handling to be triggered, depending on the state of the clustered object. The mechanism used to accomplish this behavior is called a *trampoline*.

To understand how the trampoline works, it is necessary to briefly overview how abstract methods are accessed in some object-oriented languages. At the time of compilation in object-oriented languages, it is not always possible to determine which method to run because of inheritance. To solve this problem, the method lookup takes place at runtime. One method of resolving a method at runtime is by using a *virtual dispatch table* (*vtable*). When a method of an object is invoked, that object has a reference to a class descriptor that contains the appropriate *vtable*. The *vtable* contains the necessary information to correctly perform a lookup.

The clustered object trampoline mechanism makes a special object, and then overrides its *vtable* with a custom version of its own creation. This new *vtable* redirects the method lookups to custom assembly code that saves the state of the registers, and notes the method number that was being called, and the table entry that was used to make the call. When a clustered object is created, its root is created and placed in the global table; however, the reference that is set in the local tables is that of this special object, not any of the reps.

When a clustered object's dereference takes place through the local tables, the trampoline code that was embedded into the special object is called. This triggers assembly code that moves the stream of execution into a special static method. This method has two pieces of information, the local table entry used to dereference, and the number of the method from the *vtable*. The local entry is then translated into the corresponding global entry, and the miss handler for that global entry is called.

2.2.5 Garbage Collection

One popular feature of modern object oriented languages is automatic reclamation of memory the program is finished with. This process of reclamation – known as *garbage collection* – removes the burden of memory management from the programmer, in exchange for some extra runtime cost. The implementation of Clustered Objects in K42 has semi automatic garbage collection functionality. Though it is important to mention this as a feature,

we do not go into detail as this feature was not central to the thesis of this work, and has yet to be implemented in our prototype.

2.2.6 K42 Clustered Objects: Implementation Details

Creating a clustered object in K42 starts with the Clustered Object inheritance hierarchy. Clients extend these base classes to create new clustered objects of their own design. Figure 2.5 shows the basis of K42's Clustered Object hierarchy, and where client classes are able to extend a base clustered object to create their own. At the top of this hierarchy are several different pre-made clustered object configurations. For example, one clustered object base class provides a fully replicated clustered object which produces one rep per processor, another base class provides a clustered object with a single rep for all the processors. We will return to this hierarchy again with Figure 4.4.

To give a better idea of what a real clustered object might look like, Figure 2.6 shows how a simple replicated clustered object works in K42. The *CounterLocalizedCO* class is a rep that provides an integer counter that has a local integer on each processor. When the counter is incremented or decremented, the local copy number is used, then when the value is called, all the reps have to be polled. This same code is reintroduced in Chapter 5 as the basis of the evaluation of our prototype. The details of this code are not critical, but this example illustrates what a simple clustered object might look like. Appendix B has more clustered object code.

2.3 The benefits of clustered objects

The clustered objects model has many benefits [3, 4, 19, 28]. This section overviews some benefits. Broadly, the benefits can be thought of as benefits for the programmer, and benefits to utilization. The following subsections lists some of the benefits the programmer experiences from using clustered objects, and lists some of the reasons clustered objects increase utilization, respectively.

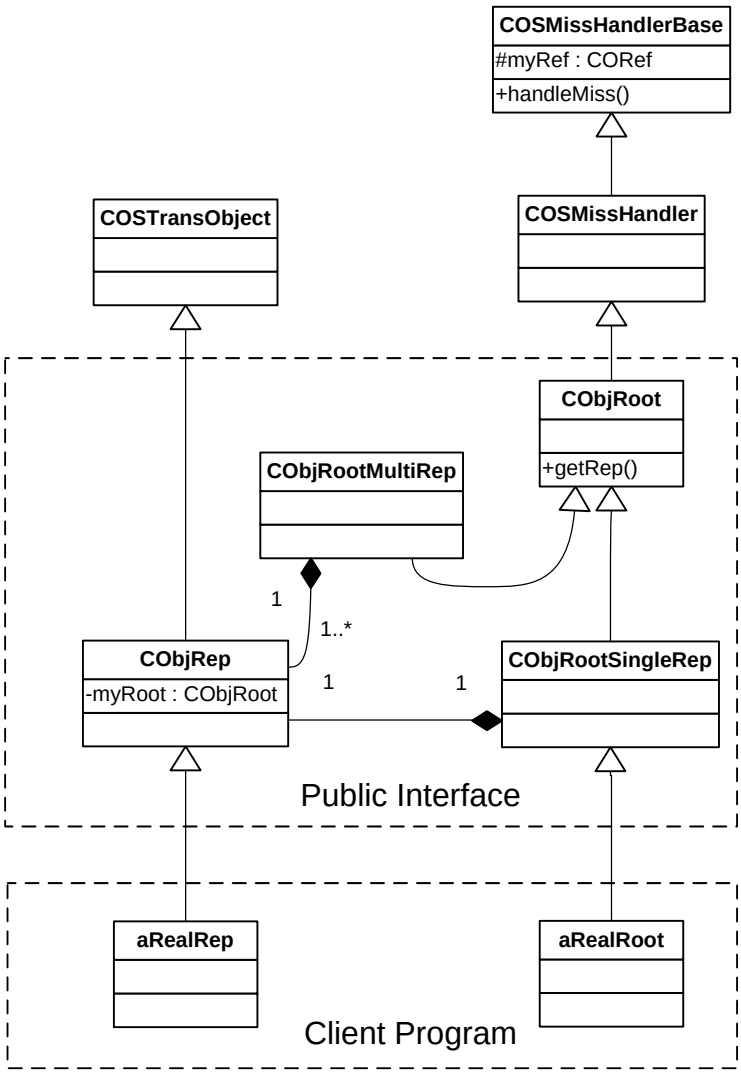

Figure 2.5. *The K42 base classes that are used to create a clustered object*

```
class CounterLocalizedCO : public integerCounter {
    int _count;
    CounterLocalizedCO() { _count = 0; }

    public:
    static integerCounterRef create() {
        return (integerCounterRef)((new
        CounterLocalizedCOMH())->ref());
    }
    virtual void value(int &val) {
        MHReplicate *mymh=(MHReplicate *)MYMHO;
        CounterLocalizedCO *rep=0;
        _count=0;
        mymh->lockReps();
        for (void *curr=mymh->nextRep(0,(ClusteredObject *&)rep);
        curr; curr=mymh->nextRep(curr,(ClusteredObject *&)rep)) {
            _count =rep->_count;
        }
        mymh->unlockReps();
    }
    virtual void increment(){ FetchAndAdd(&(_count),1); }
    virtual void decrement(){ FetchAndAdd(&(_count),-1); }
};
```

Figure 2.6. *An example of a replicated clustered object.*

2.3.1 Programming Benefits

Some of the benefits experienced by the programmer include:

Ease of use encouraged by:

1. Clustered objects reduce the use of ad hoc mechanisms for increasing locality. Our experience has shown us that without some underlying model like clustered objects it is hard for programmers to consistently and correctly apply these optimizations.
2. For the programmer, accessing a clustered object is no harder than a regular object access.
3. The assisted destruction of a clustered object simplifies the deactivation and removal of a clustered object.
4. Clustered objects do not need existence locks, a clustered object can be accessed any time. This also increases utilization by avoiding locking overhead.
5. Incremental optimizations. Initially a clustered object can consist of just a single rep, this is logically the same as a non clustered object implementation of an object. If or when in the systems evolution the clustered object becomes more contended, a distributed or partially distributed implementation can be swapped in without any changes to the client code.

Linguistic features supported:

1. Clustered objects preserve the strong interfaces essential to good object-oriented design.
2. Clustered objects are type safe.
3. Clients need not concern themselves with the internal structure of the clustered objects; neither the location nor the organization of the reps affect what the client sees.

2.3.2 Utilization

Some of the reasons that clustered objects promote better utilization:

they enable utilization by:

1. The structure provided by clustered objects facilitates the type of optimizations normally applied to reduce sharing and gain better multiprocessor performance and scalability. These include replication, migration, partitioning, and locking.
2. Furthermore, the process of changing the internal structure of a clustered object can even be done dynamically for systems that need to accommodate a varying workloads of requests.
3. Similarly, several different implementations of the same clustered object with the same interface can be present in the system at one time. Each of these implementations can be optimized for a different usage pattern.

they help efficiency by:

1. The time overhead incurred for accessing a clustered object is small (one extra MIPS instruction [19]); however, clustered object creation does have a higher overhead than regular object creation.
2. Clustered objects optionally support lazy creation of reps, so memory is not wasted on unused reps.

2.4 Summary

This chapter has presented a brief summary of the clustered object model. Clustered objects are a model of partitioned objects. The reason that clustered objects partition their objects is to promote the use of the de-clustering strategy which is a proven method of increasing locality in OSes. The end result of this strategy is increased utilization.

A clustered object is composed of a single root and one or more reps. Reps are the primary unit of distribution across processors, and act as the local proxies for the entire

Benefit	
Programming:	
ease of use	promotes optimization reduces ad hoc mechanisms easy access easy destruction no existence locks easy incremental optimization
language features	strong interface type safe oblivious clients
Utilization:	
promotes utilization	promotes optimization easy incremental optimization dynamic adaptation multiple implementations
efficiency	minimal access overhead no existence locks lazy creation of reps

Table 2.1. *Summary of the benefits of using clustered objects*

clustered object. When there is no local rep, a miss handler object is consulted to locate or create a new local rep. This forms the basis for the clustered object lazy initialization of reps. A combination of the clustered object's ID and reference are what is used to populate local and global tables to allow access to the reps. Once these tables are populated, accessing a clustered object requires only one instruction more work than accessing a regular object.

Using clustered objects has many benefits. To summarize these benefits, Table 2.1 lists them.

Though there are numerous benefits from clustered objects, they have not really been adopted by many programmers. This is because, up until now, clustered objects have not been able to be used anywhere outside of K42 and its predecessors. The next chapter details our solution to this situation.

Chapter 3

Challenges in Building a New Clustered Object Library: Dependencies and Constraints

Clustered Objects have shown their value in the concurrent OS K42 [4]. There are also other clustered object runtime facilities already in research OSes [6, 19] and there are possible plans to integrate clustered object style systems into other systems; however, none of these are in systems that could be considered in mainstream use and accessible to the average programmer.

The clustered object model as introduced in Section 2.1.3 is a general model which could be implemented in many different ways. The model itself has no relation to a particular OS. Furthermore the model has no direct relation to an OS kernel.

This chapter explains the dependencies of previous clustered object systems in the context of an attempt to take the Clustered Object facility from the K42 OS's kernel and move it into a user-level library [35] for the *Linux* [9] OS. Linux is a popular open source operation system originally written by Linus B. Torvalds [9]. We call this prototype Scalable Clustered Objects with Portable Events or *SCOPE*. Specifically, SCOPE aims to transplant the Clustered Object facility that was heavily integrated with the K42 kernel and export it into a more portable library for Linux applications. This library would have a minimal dependence on the underlying OS; hence, it would be more portable. The result would allow

Linux programs to load the SCOPE facility at runtime, and then use its features much like a thread library, such as the *pthreads* [23] library.

This chapter details the challenges we faced in the design of SCOPE. First, we describe the different delivery modes for the clustered object service, then we visit the dependencies that previous clustered object implementations have had on the OSes they were built for.

3.1 Leaving the Kernel

A reimplementation of Clustered Objects for Linux could have taken many different forms. Linux is open source, so development could have taken place anywhere from the Linux kernel to user-level. This section considers these options and describes the advantages that the library option has over all the others.

3.1.1 What are the options?

In the Linux environment, there are four places we consider that the clustered objects model could have been re-implemented.

The Linux kernel: the Clustered Objects model could have been implemented — as it has been in the past [6, 19] — as a part of an OS kernel. This would allow for unrestricted access to the system [12]; however, the maintenance of a kernel patch (which is how most extensions to the Linux kernel are provided) can be hard [10, 15, 16], and any bugs caused by the implementation could easy crash the entire system instead of the process which clustered objects are being used in [35]. Besides those two development problems, we feel, a kernel patch would hinder adoption as most Linux users don't know how to correctly recompile their kernel from its source code.

A Linux loadable kernel module: Linux allows for dynamically loadable kernel modules [12, 21]. These modules have access to most kernel resources, and do not have to be compiled into the kernel. Kernel modules have less access to the kernel, so they

are less able to cause a system crash. Additionally, kernel modules are dynamically loadable, so the kernel does not have to be recompiled (and the system restarted each time), this would ease development and would increase adoption [21].

A Linux source code library is the simplest option of these four. The clustered object source code is just imported into the user's program and compiled. This is how some simple systems like unit testing frameworks are executed. Of course in a user-level program, the SCOPE code has no direct access to the kernel except through the system call interface, but the user code has easy access to the library. One downside to this option is that if there is more than one program using the library, the code would be duplicated. This would also lead to maintenance problems as the library code in each program would have to be updated separately.

A Linux user-level shared library is the most enticing of the four. A library has all the benefits of the user-level program, without the disadvantage mentioned above. The library code is compiled in a special manner into a separate executable file. This file is then integrated into the client program when the client program loads [35]. The library code is shared amongst client applications, and Linux's library versioning methodology allows multiple versions of a library to coexist [35]. The user-level library also has the advantage of being the most portable between (multiple versions of Linux and possibly even other) OSes. Unlike a kernel patch and loadable kernel module, a user-level shared library does have the disadvantage of restricted access to the kernel through the kernel's system call interface.

3.2 Dependencies

A Linux user-level shared library is an ideal solution in this case; however, since all other clustered object implementations have been in OS kernels, it is of value to see what kernel resources they relied on. In the predecessor to K42, Tornado, several things were necessary [3]. Tornado's concurrency is fundamentally necessary, as well as three kernel facil-

ities: object translation for looking up clustered objects, memory allocation for acquiring memory that will not cause sharing, and IPC facilities for triggering execution on other processors are all facilities were used for previous clustered objects implementations [3]. Special error handling mechanisms are also used to achieve efficient and robust operation. The remainder of this Chapter describes in more detail how these dependencies play out in the clustered object implementation.

3.2.1 Concurrency

It is obvious that clustered objects require concurrency. In Tornado, there were threads of kernel execution. Beyond simple multitasking, locking/synchronization, and atomic operations are required. Tornado had all these synchronization mechanisms.

3.2.2 The Object Translation Facility

When a clustered object is accessed on any given processor, the *Object Translation Facility* in Tornado is used to locate the clustered object's representative on that specific processor. This is implemented as two sets of tables per address space. The tables are: a global table of pointers to a per clustered object management object, and a per processor table of representatives. When there is no representative listed in the processor table, the global table is consulted for a management object which manages all instances of clustered objects [3, 19]. These tables illustrated in Figure 2.2, form the basic mechanism that allow clustered objects to be accessed and allow lazy representative creation.

3.2.3 Kernel Memory Allocation

The *Kernel Memory Allocation* (KMA) is a cache aware allocator facility manages the free pool of global and per processor memory. It can allocate memory pages that are local to a target processor.

On certain machines, the access times for different parts of memory by different processors can vary. This phenomenon is known as non-uniform memory access or NUMA. Many large SMP computers experience strong NUMA effects, and some new multi-core processors also experience a NUMA-like effect due to the way they access the computer's memory bus.

In the Clustered Objects facility in Tornado, the C++ *new* operator is overloaded to use the facilities of the KMA to ensure that the default memory allocation behavior is to allocate memory from the processor that made the call and that the allocated memory is padded to the nearest cache line when necessary [3]. The location of the memory allocation close to the processor reduces NUMA effects during accesses and the padding has the effect of stopping false sharing. On non-NUMA machines, the location of the memory allocation is not of great concern; however, the padding still is.

3.2.4 Protected Procedure Call

The *Protected Procedure Call* (PPC) facility is Tornado's IPC mechanism. Protected Procedure Calls allow for two things: a processor to make a call to a clustered object in a different processor's and memory space, and allows a processor to invoke a method to be run on a different processor in the same memory space. Tornado is a micro kernel style OS [19, 34], so most of the messages that are passed between servers are done via the PPC mechanism; hence, PPC is heavily performance optimized and generally considered a lightweight mechanism [19].

In the context of past clustered objects implementations, PPC performed an interesting function: it was used to perform operations on large data sets remotely. If some data is already in the cache of the remote processor, this can save cache misses on both processors. In Tornado, the PPC mechanism only takes the runtime equivalent of a few tens of cache misses to execute [19], so it can be used in any case where a local operation would take more cache misses than that.

3.2.5 Error Handling

As an OS must be resilient, there is error handling built into K42 and Tornado. In K42 there is a system type called a `SysStatus`. The `SysStatus` is a 64-bit value that is passed back from almost every function call in the system, and regular return values are passed as reference parameters of the functions. This convention is used in preference to exception handling mechanisms in C++ for efficiency and customization reasons. Exceptions have been shown to have some overhead associated with their use, so the K42 developers did not want to use them in their kernel code. `SysStatus`es can be created by low level interrupt handlers, a venue where exceptions cannot be used effectively.

3.3 Summary

The Linux OS provides a wide variety of delivery methods for the clustered object facility. Of these, a Linux shared library encourages portability by being less dependant on the system, but still provides features like versioning that would make SCOPE easier to develop and manage.

However, in a Linux shared library, the library code does not have direct access to kernel resources, but rather it has to use the same system call interface that all other user-level programs do. Without direct access to the kernel's resources, SCOPE is not able to work like previous clustered object implementations. These dependencies are an indication of the challenge in moving Clustered Objects out of the K42 kernel. Since some of these dependencies are for systems traditionally found only in an OS kernel, reproducing them at the Linux user-level presents some interesting design challenges. The next chapter details some of our design decisions and the solutions we found for these dependencies that allows SCOPE to function at Linux user-level.

Chapter 4

SCOPE: Prototype Design and Implementation

All of the mechanisms described in Section 3.2 and summarized in Table 4 below are possibly necessary to provide the functionality of Tornado's and K42's Clustered Object system; hence, some suitable equivalent had to be found for user-level in Linux. This chapter details how the design of SCOPE tries to satisfy these dependencies, the design decisions we made and some of the incremental process we used to make a user-level library out of the K42 implementation of Clustered Objects. This design present in this chapter is this work's major contribution to this area. Mainly the design of a user-level clustered object facility that has a low coupling with the underlying OS.

This chapter begins by showing how we satisfied each of the five dependencies mention in Section 3.2. Section 4.1 shows how *pthreads* can be used as a concurrency model. Section 4.2 shows how we reproduce object translation with our *portability events*. Sec-

Dependencies	**Function in K42/Tornado**
Concurrency	need a model of concurrent execution
Object Translation	necessary to perform a clustered object lookup
Memory Allocation	necessary to supply padded and pooled memory
Protected Procedure Call	necessary for providing bulk remote data processing
Error Handling	necessary for efficiently reporting low level errors

Table 4.1. *The dependencies the Clustered Object facility had in Tornado and K42, and a brief summary of their purposes*

tion 4.3 discusses reproducing kernel style memory allocation. Finally, Sections 4.4 and 4.5 describe how we reproduced the PPC IPC mechanism and the error handling mechanisms were reproduced. This chapter concludes with a description of the iterative implementation process employed during the development of SCOPE.

4.1 Concurrency

As discussed in Section 3.2, clustered objects need some sort of concurrent execution model upon which to work. Concurrency can be reproduced in the Linux user-level with the *pthreads* [23] library.

In K42's Clustered Objects, the Virtual Processor (VP) abstraction is the level at which per processor information is maintained, and ultimately the kernel-level unit of concurrency. VPs are thus the logical entities to which representatives are assigned, and each VP has its own local lookup table. VPs are assumed to map closely to the physical processors of the machine.

One challenging decision that needed to be made in the design of SCOPE was to devise a strategy for mapping VPs to the *pthread* notion of threads. The *pthread* library does not guarantee such a close mapping of threads to processors; however, *pthreads* allow the programmer to set an attribute that maps newly created threads directly to *system* threads. System threads are treated as processes in the Linux scheduler, but Linux provides no guarantee of the mapping of processes to processors in the system. This means that:

1. threads might not be running on every processor,
2. there might be more than one thread running on the same processor, or
3. a thread could be moved from one processor to another without notice.

All three of these situations are potentially problematic, but we solve these problems by creating a 1:1 mapping of one *pthread* to one K42 VP.

Mapping each thread to a VP means that each thread will have its own lookup table

and representatives. The lose mapping of threads to processors does not pose a correctness problem as each thread has its own per thread data instead of what had originally been per processor data. When two threads run on different processors, this mapping is no different than in K42. If a thread moves from one processor to another, it will suffer the initial cache misses to move its local table and representatives with it, but then will continue to operate as normal. Unfortunately however, when two threads run on the same processor (one after the other), they will use separate representatives. Though this is potentially non-optimal as they could both be using one representative and they may also expel each other's cache data as they operate on the same cache, it is not incorrect.

This mapping also avoids some other problems: if more than one thread mapped to a VP, it can no longer be assumed that representatives are being used by only one thread. This means other measures have to be employed to ensure local data consistency and guard critical sections. Furthermore, doing a lookup requires a critical section (to ensure that the process is not moved during the lookup) and the ability to accurately and quickly find the number of the processor the thread is running on.

Though this mapping solves many problems, it has at least one flaw. To ensure concurrency in a system, threads are often liberally used. In this model, each thread could have a representative for each clustered object and a local lookup table. Since the representatives are created lazily, they will not waste memory when not used; however, operations that visit each representative will take longer as there may be more representatives in the system. The local lookup tables present more of a problem. Each table has to have an entry for each clustered object in the system. If there are T threads and C clustered objects in the system with table entries of size S, the size of the tables in the system are at least TCS. In a system with lots of clustered objects, and lots of threads, the scalability of the clustered object system itself comes into question. In K42 they take advantage of their 64-bit memory space, and the sparse nature of these tables to allow the virtual memory system to handle theses tables, and not waste memory on unused table entries. However in the current implementation of SCOPE, these tables have a default value written to them, and

so the full size of the tables is stored in memory. This is a huge source wasted memory, which needs to be addressed in future work.

Like the K42 version of Clustered Objects, SCOPE allows for a *degree of clustering*. The degree of clustering of a K42 Clustered Object is the number of VPs that are assigned to a single representative. It may be possible to accomplish this same degree of clustering in SCOPE by making threads that are known to run on the same processor map to the same representative, but more work needs to be done to see if this is actually feasible to find and maintain a mapping like this. The challenge in establishing a mapping like this is actually figuring out which threads are going to be on which processor, this is information that user-level programs are generally not privy to. Furthermore, the system scheduler could move a thread from one processor to another at any time, in which case parts of the mapping would have to be invalidated, then reestablished. Processor affinity settings could be used to minimize thread moves, but this potentially limits many of the benefit of having movable threads in the first place.

We believe this 1:1 mapping is as close to the K42 VP abstraction as can be easily accomplished in *pthreads*, and provides the concurrency necessary to run a general implementation of clustered objects. *Pthreads* also provides the mutexes, semaphores, and barriers which are used for locking and synchronization in SCOPE [14].

Finally, to implement atomic operations we are able to embed custom platform specific assembly code into the source code; however, since Tornado and K42 were designed for special hardware[1], we are not able to reuse any of their atomic assembly code. To interface with the remaining K42 code in SCOPE, wrappers are needed to wrap things like the K42 *Lock* class with a *pthread* based implementation.

Some of the simple operations that were used in the K42 Clustered Objects facility, and hence were needed for SCOPE were an atomic *FetchAndAdd* which atomically takes a value from memory and adds a value from a register to it. Another important atomic

[1]Tornado was written to run on MIPS processors, and K42 on PowerPC 64-bit processors. K42 was initially supported on X86 32-bit and PowerPC 32-bit as well, but those versions were deprecated and then eventually removed.

operator is *CompareAndSwap* which compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value. *CompareAndSwap* can be used to implement synchronization mechanisms. Equivalent versions of all the K42 PowerPC atomic assembly routines had to be recreated.

4.2 Object Translation Facility

There is no equivalent to the Object Translation Facility provided in the Linux user-level. As the dispatch that is used in the K42 Clustered Objects is heavily integrated with the rest of the Clustered Objects facility, we took the simple approach of using the K42 Object Translation facility in SCOPE. The Object Translation facility had to be changed slightly though. A fundamental difference between the Tornado/K42 kernel and Linux user-level is the inability to map real addresses to different physical address on a per processor basis. The K42 virtual memory system maps addresses on a per processor basis, so the local representative lookup table could be located at the same virtual address for each processor, but be backed by a different physical memory and therefore a different table. This functionality is allows for a quick clustered object lookup process, without the need to find a per processor table of representatives.

The remainder of this section provides details about the translation facility. Section 4.2.1 introduces our new DREF preprocessor macro that allows us to perform a lookup in SCOPE, then Section 4.2.2 introduces our portable event structure, and Section 4.2.3 evaluates the benefits of this new structure.

4.2.1 DREF: the Dereferencing Macro

The use of this local table means the dereference of a clustered object only takes two regular pointer dereferences. A reference to a clustered object in K42 (called a CORef) is a pointer into this local table. In the code that access a clustered object, that makes an access of a clustered object *c* with method *m* looks like ***c->m()*; however, the K42

Clustered Objects implementers realized that this was implementation dependant so they created a preprocessor macro called *DREF* which performs a clustered object dereference, so a typical clustered object access in K42 actually looks like: *DREF(c)->m()*. Of course, *DREF* is just defined to prepend the **.

The use of the *DREF* macro eases our implementation because we can define it to mean whatever we need, so we are not limited to the ** straight dereference. Redefining the *DREF* can give us a way to make up for the lack of per processor memory. In a newly defined *DREF* we can look up which processor we are on, then access the appropriate local table.

There are several ways that we could find which processor we are executing on:

- the X86 *cpuid* assembly instruction may be able to be used; however, this instruction has the side effect of filling all the registers and flushing the processor's pipeline.
- A Linux system call could be used to find out the current CPU that a thread is running on. Unfortunately, system calls are a very high overhead mechanism, so they are not considered further.
- If we make a simple assumption that each thread is running on a different processor (the default for *pthreads* in Linux and can be easily set if not), then we can just use information stored in the thread stack. In *pthreads*, the *pthread_key* functionality allows you to store a named piece of data that is specific to each thread. For example, we could create a key to represent the VP that we designate the current thread to be running on. We could even completely remove our assumption if we used Linux's scheduling API to manually assign threads to processors, although this removes some of the benefit of having kernel level threads in Linux.

4.2.2 Portability with Events: START_EVENT and END_EVENT

To enhance the portability of SCOPE, we introduce the notion of a *clustered object event*. An event is an environment in which a program can make calls to clustered objects. For the

```
if (pthread_getspecific (*getVpKey ()) == NULL) {
  VPNum *vp = assignVP ();
  pthread_setspecific (*getVpKey (), (void *)vp);
  unsigned long *offset = (unsigned long *) malloc(
                                          sizeof (unsigned long));
  *offset = (*vp*sizeof(LTransEntry)*NUMBER_OF_CLUSTERED_OBJECTS)
                                          /sizeof(unsigned long);
  pthread_setspecific(*getLTransKey (),(void *) offset);
}
pthread_setspecific (*getGenRecKey (), (void *) activate ());
```

Figure 4.1. *The SCOPE* `START_EVENT` *macro. On the first run, a thread specific key is set to the base of the thread's local table and A VP number is assigned to the thread.*

duration of an event, all the information necessary to complete a dereference is available. The event is responsible for preparing[2] the current calling thread, and for triggering any necessary cleanup[3] after an event is over.

Starting an event is simple, the client program calls the *START_EVENT* macro. The *START_EVENT* macro is the system specific macro that will set up the current thread for a clustered object dereference. As shown in Figure 4.1, in our current *pthreads* implementation, the macro checks the *pthread_key* values to ensure they are initialized, and in future versions will check if a representative migration to another processor has been triggered and act accordingly, and record garbage collection information (an active event count to detect quiescent states).

After a clustered object dereference has happened, the client program should call the *END_EVENT* macro. Like the start event macro, this is a system specific macro. *END_EVENT* marks the end of an event. In the case of our current *pthreads* implementation, the macro does nothing; but, in a future version it will record garbage collection information, then

[2]In SCOPE to prepare a thread the current CPU that the thread is running on has to be resolved, the base pointer to the current CPU's local translation table has to be acquired, and a pointer to the current generation record has to be established (if garbage collection is active).

[3]Cleanup entails checking if any dynamic operations are needed like a migration, and triggering any necessary garbage collection statistics to be updated.

```
#define PTHREAD_DREF(ref) (*(ref + *PTHREAD_LTRANS_OFFSET))

#define PTHREAD_LTRANS_OFFSET   \\
   ((unsigned long*)pthread_getspecific(*COSMgr::getLTransKey()))
```

Figure 4.2. *The SCOPE* `DREF` *macro. A thread specific key is used to get to the local table, then an offset (which is the reference passed into* `DREF`*) is added to find the table entry.*

```
CORef ref;
ClusteredObject::Create(ref);
//In K42
DREF(ref)->someMethod();

//vs. in SCOPE
START_EVENT;
DREF(ref)->someMethod();
END_EVENT;
```

Figure 4.3. *An example of how a clustered object is called.*

if necessary trigger clustered object deallocations. Figure 4.3 is a code example of how a clustered object is accessed in K42 and with event macros.

When SCOPE is ported back to K42 or other systems like Microsoft Windows or a Java virtual machine like IBM's J9, these macros can be changed. For instance in K42, *START_EVENT* and *END_EVENT* could be empty because in K42 all of the features that are supported do not need to be triggered by the client program.

4.2.3 Access Patterns and Portability

The goal of these events is to increase the portability of SCOPE and clustered objects written for SCOPE. The event model makes it simple for a clustered object written on one system, for example the preexisting set of K42 clustered objects, to run on other system (such as SCOPE's *pthread* version).

With the three macros discussed in the sections above, we can see what the standard access of a clustered object looks like in SCOPE. First, a *START_EVENT* is called, then the *DREF*, then the *END_EVENT*.

All of that said, if the client programmer knows which system the clustered object is intended for, and how that system works, they can opt to only call the event macros when they are actually needed. In the case of the current *pthread* version, it would suffice to call the *START_EVENT* only once when a new thread is created; however, doing so is not portable to other systems, or possible later versions of the *pthreaded* system.

4.3 Kernel Memory Allocation Facility

Although we can attempt to reproduce some subset of the functionality associated with the kernel allocation facility, the custom memory allocation available in the K42 and Tornado kernel are hard to completely reproduce at Linux user-level. In SCOPE, we need a padded allocator to ensure that consecutively allocated small objects (smaller than a cache line) do not experience the false sharing mentioned in Section 1.2.2 because they end up residing on the same cache line.

To reproduce padded allocation, we decided to use a function wrapper for the *malloc* function which is used to obtain new free memory from Linux. Extra memory is allocated by the *malloc* wrapper to ensure that the memory block returned to the caller is aligned to the nearest cache boundary. We do not dynamically determine cache line sizes, but allow it to be set at compile time as a constant. Because the Linux paging system is controlling the memory behind what we allocate, we cannot assume a linear mapping of addresses to memory; however, because page sizes are multiples of the cache line size, our padded allocator should still not allow consecutively allocated memory to end up on the same cache line. Pooled allocators (which create a pool of memory for each processor then allocate from that pool) are also found in K42. SCOPE could benefit from a pooled allocator, but at this time, one has not been implemented.

4.4 Protected Procedure Call Facility

Applications that use SCOPE are intended to be run in a shared memory environment, so there is no need for IPC to be used to communicate inside SCOPE. Since the PPC facility's benefits are for only a certain small set of cases as outlined in Section 3.2.4, and its implementation would be quite complex, PPC has been omitted from the SCOPE prototype.

4.5 Error Handling

To reproduce the error handling functionality of the K42 in SCOPE we introduce the notion of a Clustered Object Return Code (CORC). The problem with the K42 `SysStatus` is that for many applications an encoded 64-bit value is too low level. The CORC is a more heavy weight mechanism that carries more information about the last error, including a possible error message, and more details about how the error occurred. These aid in debugging SCOPE and allow for more complete error reporting to the client application.

For example, when an error is triggered in K42, all the client sees is an error code and module code which identifies where the error was triggered. If the error information is published in the K42 error lookup system, the client can then look up what that error code means, and cross reference with the module where it was triggered. In SCOPE the error message, not only holds the K42 error information, but the file and line number where the error was triggered, a plain English description of the error, and then any extra information about the error that might be necessary.

4.6 Implementation Process

The majority of the code for SCOPE was taken directly from the Clustered Object facility in K42; however, there are major differences in how SCOPE is implemented compared to Clustered Objects. An iterative process was used to incrementally move parts of Clustered

Objects into SCOPE. First, a central class or classes was chosen, and imported, then everything that those classes relied on is imported. At some point, the process reaches the interfaces that Clustered Objects uses to access other subsystems, at that point the appropriate stubs were created. Much of the functionality was removed from the classes to simplify the importation process.

Figure 4.4 is an extension of Figure 2.5 showing how we implemented SCOPE. When importing these classes, first we created a *CObjNullRep* class which was a completely empty clustered object (just a root, no reps), this is represented as everything above the mid line in Figure 4.4. As the state of the implementation advanced, we added a *CObjRootSingleRep* which is a clustered object with a root and a single representative. This was the second stage of the implementation. The *CObjRootSingleRep* was the first real clustered object that was supported; finally, once *CObjRootSingleRep* was working, *CObjRootMultiRep* was imported. This was the third and final stage. *CObjRootMultiRep* is a fully distributed clustered object with one representative per VP, it is the most complex clustered object type to support, which is why it was saved for last.

The second phase of the importation focused on the basis of the Clustered Object runtime infrastructure, the *COSMgr*. Figure 4.5 shows the *COSMgr* hierarchy and some of the classes that it relies on. The design of the COSMgr is taken directly from the K42 Clustered Object facility. The *COSMgr* is responsible for keeping track of VPs (in VP sets), and creating and maintaining all of the clustered object lookup tables (the Translation Entries or *TransEntry*). The *COSMgr* makes use of several machine specific atomics, the system scheduler, and the system's memory allocation. For creating complex clustered objects there are a set of Factory classes that can aid the programmer, and for important clustered objects (like the *COSMgr*) reservations can be made in the translation tables with the *COGlobals* class. To facilitate the lookup process, *COVTable* overwrites clustered object's initial vtables with that of the *DefaultObject*'s. In the K42 version and future versions of SCOPE, garbage collection is also done in the *COSMgr*.

The native K42 version of the *COSMgr* is a multiple representative clustered object.

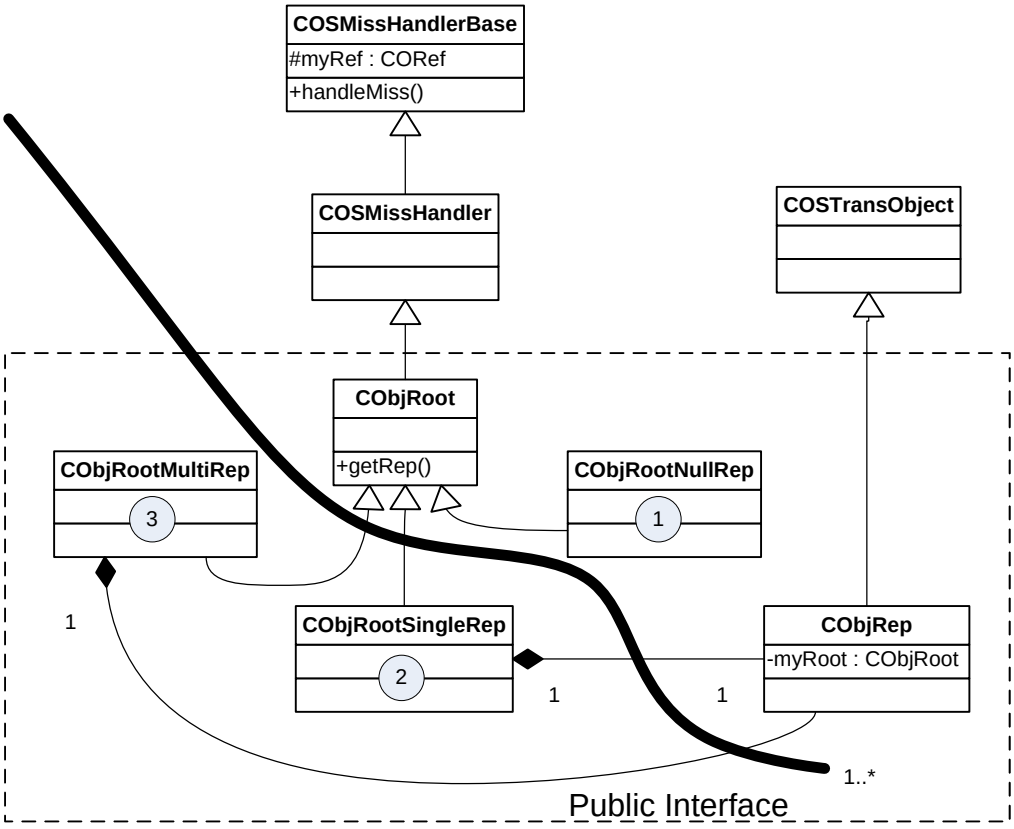

Figure 4.4. *The three stage implementation of SCOPE. In the first stage we imported everything above the thick middle line. Then, we imported the SingleRep and then the MultiRep classes and functionality.*

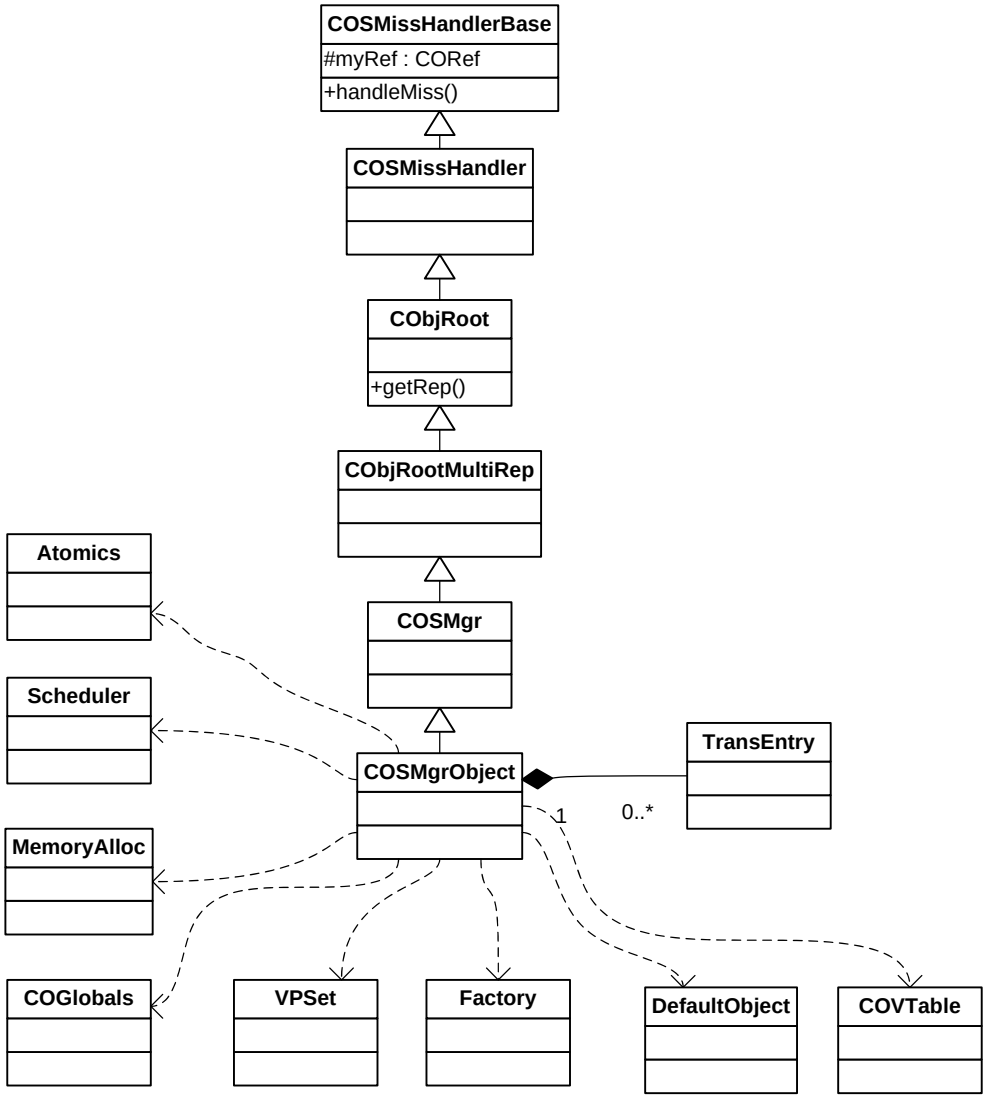

Figure 4.5. *The* COSMgrObject *is a clustered object that organizes the runtime system. It relies on many other classes in the system.*

Obviously this could not be used until the multiple rep code was working; so, to simplify our implementation we created a single representative version of the *COSMgr* to be used with SCOPE. The single representative version will not be able to handle as large a load during concurrent clustered object creation as a multi-rep implementation, but the access performance of the clustered objects is not affected by this choice.

As SCOPE evolved, so did our target environment. At first, SCOPE and its test programs were just compiled as a simple Linux user-level program as mentioned in Section 3.1.1; but as SCOPE evolved, we shifted SCOPE into a Linux shared library. This shift allowed us to have more test programs, and would allow an actual client program to start using SCOPE.

4.7 Summary

In this chapter we detailed the design decisions we made in the initial implementation of SCOPE, and we have shown how we reproduced the five dependencies previous implementations had on kernel services. This resulted in the creation of our portability events. These events allow us to correctly set up the runtime environment without the aid of custom OS code. Finally we describe how we implemented SCOPE with an iterative process to satisfy the complex interdependencies of the previous system.

The design decision presented above allowed us to implement a functional clustered object system at Linux user-level; however, these decisions also fundamentally alter the way a clustered object is accessed. The next chapter evaluates SCOPE to see how it performs both in terms of overheads and in terms of the benefits mentioned in Section 2.3.

Chapter 5

Testing and Validation

This chapter presents an initial evaluation of SCOPE. Here, we attempt to validate if, in the face of the system level complexities introduced by sharing, a user-level abstraction can increase utilization without control of the OS. First, in Section 5.1, we verify some of the assumptions that the library was built upon. Then in Section 5.2 we evaluate the performance of the new library to see if it can provide benefit to a host system. Finally in Section 5.3 we discuss the benefits outlined in Section 2.3 to establish whether SCOPE supplies these same benefits.

5.1 Evaluating Assumptions

This section details the assumptions we made when designing SCOPE. The intent of clustered objects is to reduce sharing, thus reducing the synchronization overhead and bus contention caused by sharing. But, is all this work to reduce sharing needed? On the specialized hardware that Tornado ran on, it has been shown that reducing sharing was advantageous [19]. It has also been shown that, on the PowerPC hardware that the K42 OS was designed for, reducing sharing is advantageous [4]. But, on a commodity X86 machine made by AMD, sharing might not have the same impact as seen on a 24 processor machine [4]. Some simple tests are needed.

In the first part of this chapter, we present a simple set of experiments in the context of an integer counter that attempts to analyze the impact of sharing on an X86 system, and

then report on the results of these experiments. The second part of this chapter presents an evaluation of the performance of SCOPE in the context of the same integer counter, and then in the context of a broader software engineering characteristics.

5.1.1 Quantifying Sharing

There are sources of variability in our test; but, these sources exist in real systems as well. Although they make getting accurate test numbers harder, they are an accurate recreation of the environment where real programs run.

Multiprocessor machines are complex, both in-terms of architecture and software. From a testing prospective, there are a lot of variables to try to control while testing for the presence of sharing. Some obvious sources of variability in the following experiments include:

- General purpose OSes (like Linux) provide no guarantee of what is running at any given time.
- We have some expectations about the hardware and how the cache works etc; but, without special hardware we are unable to actually see what is going on inside our test systems.
- Other things can be happening in hardware/memory, i.e. device interrupts, DMA etc that may pollute the cache or flood the memory bus.

Some of these variables are not things we can easily control in a simple test, so we hope to minimize anomalies by running larger data sets. This test is intended to provide an indicator that sharing is occurring, and a rough estimate of the relative performance when sharing is involved in some simple test cases.

5.1.2 The Integer Counter Example

One of the simplest examples used to evaluate other clustered object systems and other concurrent systems has been an integer counter [3, 31]. An integer counter is a class with a single integer field. The class has an interface which consists of three methods:

inc add one to the this counter

dec subtract one from this counter

value get the current value of this counter

In C++ the integer counter interface would have a definition like this:

```
class IntegerCounter{
public:
    virtual void inc()  =0;
    virtual void dec()  =0;
    virtual int value() =0;
};
```

One can imagine a class like this being used to count frequent events. The design of these experiments needs to ensure events are frequent enough on each processor that they create a noticeable amount of overhead from sharing. On our simple test system, we intentionally try to interleave these calls between processors; in part, that is because our test system only has two processors. On a system with more processors there are more potential streams of execution that can contend any one cache line, and cache synchronization can cause more overhead.

5.1.3 Experiment 1: Creating Contended Counters

This first experiment will test the integer counter in different configurations. We first try to validate that sharing is indeed occurring in the system. Then, we validate whether some of the techniques used in the clustered object library are in fact able to reduce the sharing experienced and therefore the overhead induced by sharing. In short, this experiment aims to show that sharing does happen, and that the techniques employed in the clustered object system do in fact reduce sharing.

5.1.4 Hardware Setup

The machine that this test was run on was a dual processor X86 based machine. Some details about the machine were collected with the CPU-Z system information tool. More details about this machine can be found in Appendix A, but briefly the machine has:

CPUs 2 AMD Athlon MP 2400+ processors,

Memory 1GB (2 x 512MB) of 133MHz DDR RAM

Cache each processor has 64kB of L1 data cache, and 256kB of L2 cache, both with 64-byte cache lines.

5.1.5 Software Setup

Our machine was running SUSE Linux 10.0, with an underlying Linux kernel version of 2.6.13SMP. This test did not use *pthreads* or the clustered object library, just Linux processes. Processes were used to keep the test as simple as possible; however, the amount of things that can be easly done in the test programs is limited without the added concurrent mechanisms provided by *pthreads*. The testing processes were run in single user mode, and the subtleties of this testing scenario are provided in Appendix C. After the initial process is started, child processes are created via the Linux *clone*[1] system call, with the VM_CLONE option enabled to set the child share the parent's memory space[2].

5.1.6 Procedure

Four different configurations of the integer counter were tested, each designed to show different characteristics. Cases 1, and 2 implement the counters as if there were no overhead caused by sharing. Cases 3 and 4 partition the counters to see if that reduces sharing. All

[1]The Linux *clone* system call creates a new child process. With the VM_CLONE flag sent, the child process will be setup to use the same memory space as the parent except with a different runtime stack.

[2]VM_CLONE is how many threading systems create their threads in Linux

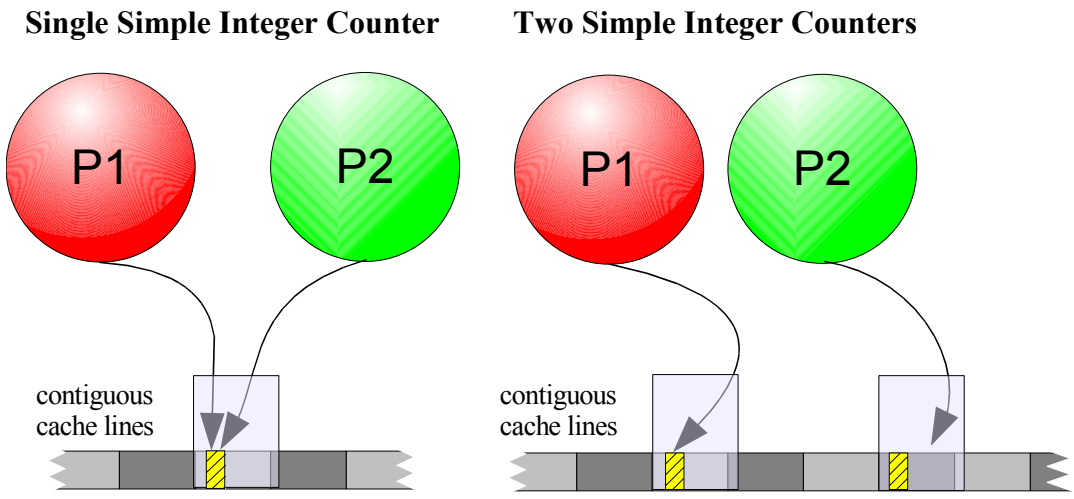

Figure 5.1. *Two configurations of the SimpleIntegerCounter and processors P1 and P2 accessing them. Each processor accesses the internal* int *field through an* `inc()` *method.*

the code for these can be found in Appendix B. The two simplest configurations are as follows and are illustrated in Figure 5.1:

Case 1: one *SimpleIntegerCounter* This test is comprised of a single *SimpleIntegerCounter* being shared by two processors. A *SimpleIntegerCounter* is a class that just has a single *int* field as its only member. The *inc*, *dec*, and *value* methods atomically increment, decrement, and return a copy of the integer field respectively. In this case we expect to see sharing, and therefore the inherent penalty associated with reduced CPU utilization.

Case 2: two *SimpleIntegerCounter* instances This test comprises two separate instances of a *SimpleIntegerCounter*. We ensure that they are not close to each other in memory (at least a cache line apart). A processor is assigned to access each simple integer counter. When we make a *value* call on either of these integer counters, we must return the sum of both integer counters' *int* fields. We expect no sharing in Case 2, and therefore no performance penalty.

The obvious problem with a single *SimpleIntegerCounter* in Case 1 is that the internal *int* field is being shared. Cases 3 and 4 avoid this sharing by using more than one *int* field internally, without having to create more than one instance of the *SimpleIntegerCounter* as in Case 2. We present two different configurations as shown in Figure 5.2:

Case 3: one *ArrayIntegerCounter* This test comprises an array based integer counter. The counter has an *int* array with one element for each processor. Like Case 2, when the *value* method is called, we must return the sum of the array elements. We expect to see no real sharing, but since the array elements probably are on the same cache line, we expect to see false sharing, and therefore poor performance.

Case 4: one *PaddedArrayIntegerCounter* This test comprises an array based integer counter like Case 3; however, each element in the array is padded, so that each element fills one cache line. Since each element has its own cache line, there is no false sharing between elements. As with the other two integer counters, when the *value* method is called, we must sum all the padded elements to return the total value. We expect to see no sharing (real or false) in this case, and therefore performance similar to the two separate *SimpleIntegerCounter* instances of Case 2 in Figure 5.1.

5.1.7 Quantifying Results

In this first test, each of the integer counters is exposed to the same number of invocations of the *inc* method from two processors concurrently. The total number of cycles to satisfy all the requests is what we consider to be the performance. The test is run a number of times to obtain an average performance for each integer counter. Appendix B has more details and the code for this test. Finally, we divide the average performance by the number of requests to the *inc* method to find the average number of cycles per request. Cycles per request is our metric, in keeping with previous work [3, 19].

This test is susceptible to the sources of variability outlined in Section 5.1.1. To try and mitigate variability, all tests are run in single user mode, which has been shown to be far

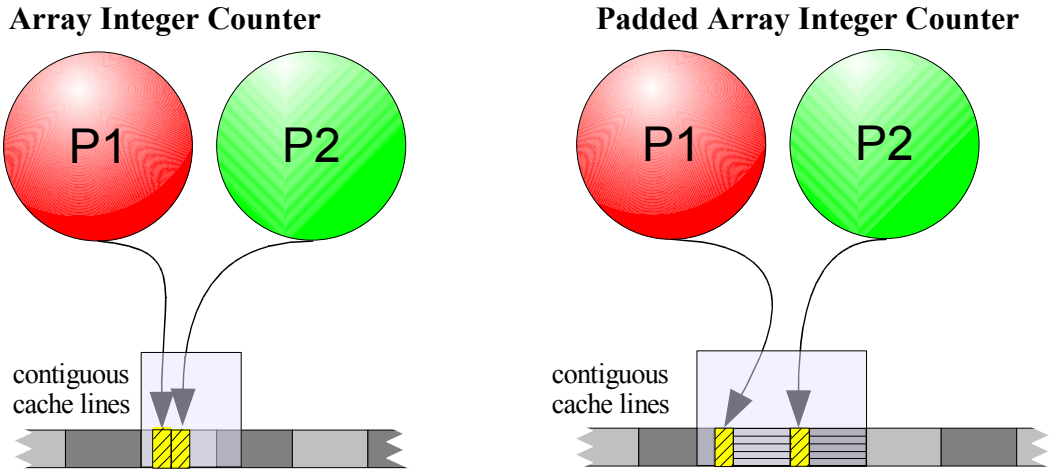

Figure 5.2. *The ArrayIntegerCounter, PaddedArrayIntegerCounter and processors P1 and P2 accessing them. Each processor accesses the internal integer array through an* `inc()` *method.*

Average cycles per inc call	
Case 1: One integer counter	453.65
Case 2: Two separate integer counters	46.35
Case 3: Array integer counter	341.72
Case 4: Padded array integer counter	48.05

Table 5.1. *Results of simple sharing test cases mentioned in Section 5.1.6*

less variable (Appendix C); however, other sources of error are introduced by unintended sharing. In section 5.1.9 we discuss these problems and how they were solved.

5.1.8 Results

Table 5.1 shows the results from this first test. Figure 5.3 shows a graph of those results indicating the average runtime for each of the test cases used in this simple evaluation. The runtimes themselves are not as important as the relative differences between them, as this shows just how much overhead sharing can cause.

As expected, the two *SimpleIntegerCounter*s in Case 2 have no memory in common,

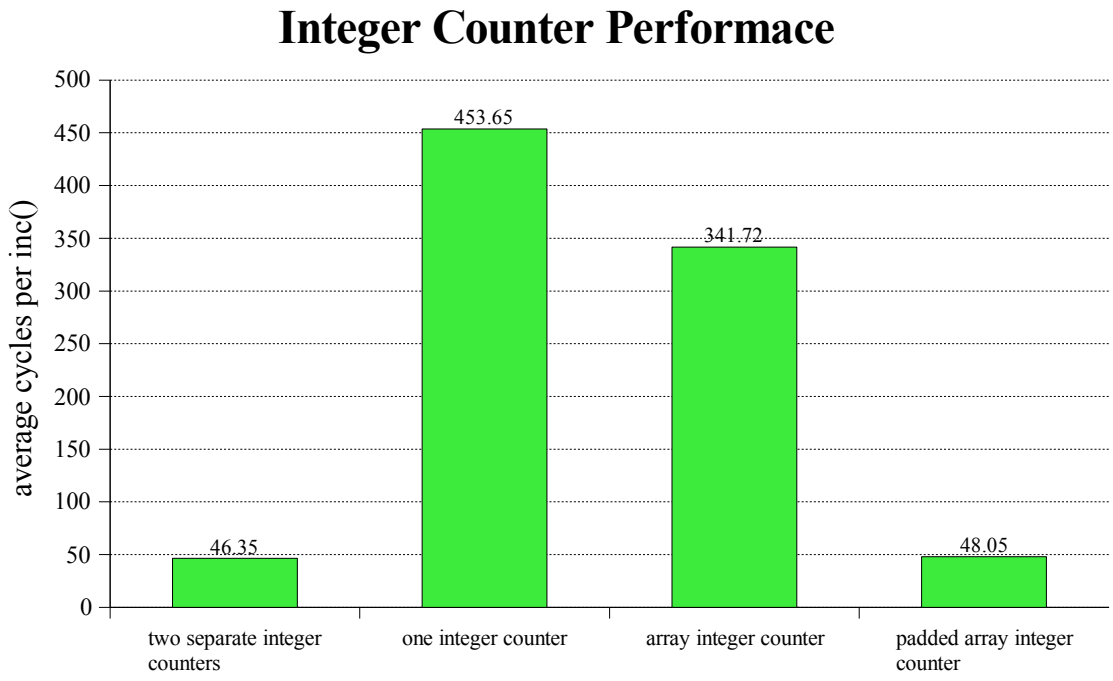

Figure 5.3. *Average runtime results of the four integer counters listed in Section 5.1.6, Table 5.1.*

hence they experience no sharing and therefore perform very well. It is interesting to note that the *PaddedArrayIntegerCounter* in Case 4 performs just as well, even though it is a single object. The counters which had real and false sharing, however, were almost an order of magnitude slower. These results indicate that there is in fact significant overhead introduced by sharing on an X86 based system; and that our assumptions are correct. Specifically, these results show that by restructuring objects in a cache conscious way, we can achieve a significant performance benefit over a poorly structured object.

5.1.9 Lessons Learned: Unanticipated Sharing

Some anecdotal evidence shows that there is a benefit obtained by using clustered objects; even when dealing with an example on this small of a scale. This is a snippet of code from the test program:

```
test(new SimpleIntegerCounter(),new SimpleIntegerCounter());
```

This code's intent was to create the two separate *SimpleIntegerCounter*s for Case 1. But, without custom memory allocation the two back to back calls to *malloc* triggered by these *new* statements may place the two *SimpleIntegerCounter*s back to back in memory. This had the unintended side effect of leaving both integer counters on the same cache line, and therefore causing false sharing.

Before it was fixed, a simple mistake like this caused an increased overhead similar to Case 3's false sharing on Case 2. If the focus of this work was not controling sharing, it is unlikely that this would have ever been noticed, while it continued to take a small toll on the system.

Of course, one ugly fix is simply to make sure the two do not end up on the same cache line:

```
IntegerCounter* c1 = new SimpleIntegerCounter();
malloc(CACHE_LINE_SIZE);
IntegerCounter* c2 = new SimpleIntegerCounter();
test(c1,c2);
```

It is important to note however that a simple constructor like Case 4's that allocates an 8 processor padded array has a hidden problem. The *PaddedArrayIntegerCounter* has a member pointer, *_val*, which points to the dynamically created padded array. *malloc* is first called during the object construction and the *_val* pointer is left in that memory, if we immediately call *malloc* again to create the padded integer array, in some cases that leaves the object's data (the *_val* pointer) and the first element of the padded array in the same cache line and hence false sharing happens between the *_val* pointer that all the processors use and the first padded element that only the first processor uses.

```
IntegerCounter::IntegerCounter(){
    _vals = (volatile int*) malloc(CACHE_LINE_SIZE*8);
}
```

Again, in an ugly fix for this, we must separate the object's data and the array:

```
IntegerCounter::IntegerCounter(){
```

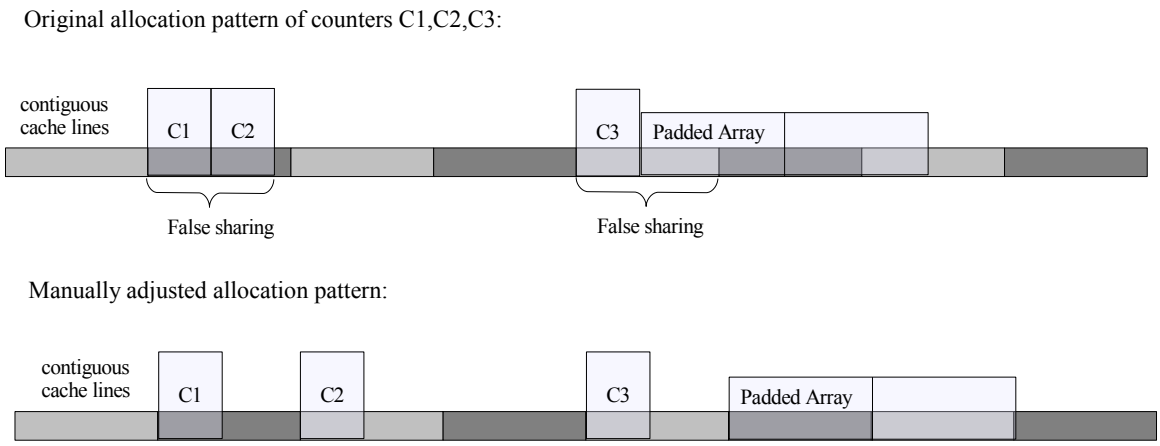

Figure 5.4. *The natural layout of the integer counters across cache lines, then the manually corrected versions. Notice how without adjustment the counters can end up on the same cache line, and therefore introduce sharing.*

```
    malloc(CACHE_LINE_SIZE);
    _vals = (volatile int*) malloc(CACHE_LINE_SIZE*8);
}
```

Figure 5.4 illustrates what is happening in the above code. It is not trivial for a programmer to take sharing into consideration when writing code. In the above examples, SCOPE would have automatically provided padded allocators thus alleviating the burden on the programmer to get these kinds of intricate details correct.

5.2 Performance of SCOPE

Now that we have shown some evidence that there can be some overhead caused by sharing in a system, we present an initial performance analysis of SCOPE to see if it can reduce the cost of sharing. This analysis is not intended as a detailed performance analysis; rather, a simple verification that SCOPE works and provides benefit. This analysis is also not intended to evaluate clustered objects as a whole, but rather just the SCOPE implementation. To that end, we are not analyzing the tradeoffs between local and aggerate operations (like `inc()` vs. `value()`), just the performance of the local operations.

For this set of experiments SCOPE is used as well as some of the counters from the previous sections; however, now the concurrency is provided by *pthreads* instead of Linux processes. The counters used from the previous sections are fundamentally the same, but differ in implementation from the previous sections to accommodate the new model. The counters should experience sharing as in the last sections, but the exact performance characteristics might be different. *Pthreads* is the de facto standard for Linux concurrent programming, so it is a reasonable model to test with.

The test framework used in these test cases is more advanced than the tests in Section 5.1.6. *Pthreads* provides extra synchronization constructs which we take advantage of to produce higher concurrency. This is accomplished by using thread barriers to align thread execution of the critical sections across each run of the test case. Because of this, the numbers produced by these test cases are more regular.

5.2.1 Experiment 2: Counting Clustered Objects

In this experiment we reexamine 3 of the 4 integer counters from 5.1.6. We do not include the two separate IntegerCounters case from above because it does not present a single interface to the client, and we have shown that our PaddedArrayIntegerCounter performs just as well.

5.2.2 Setup

The hardware setup for this experiment is exactly the same as in 5.1.4, the software setup is similar to that in 5.1.5 except we are now using *pthreads* instead of the `clone()` system call.

5.2.3 Procedure

In this experiment six different IntegerCounter test cases are examined:

Cases 1, 2 and 3: are one *SimpleIntegerCounter*, one ArrayIntegerCounter, and one PaddedArrayIntegerCounter. Cases 1-3 are re-implementations of the counters from Section 5.1.6 redesigned to work under *pthreads*. So that the tests can determine which processor they are running on, we allow them to use the MYVP macro from SCOPE. This macro provides the number of the current processor that the thread is running on. This is what is used to index into the per processor arrays.

Case 4: is one SharedIntegerCounter with a special DREF. This test case has a SharedIntegerCounter clustered object; however, we do not use the clustered object dereference mechanism, rather the regular C++ mechanism. The clustered object version of the IntegerCounter is more complex because of inheritance and the class is much larger than that of the previous cases. A replicated clustered object could not work like this, but we can still directly reference a clustered object with a single representative. We expect this to perform poorly as it will have the effects from sharing, and the virtual dispatch.

Case 5: is one SharedIntegerCounter clustered object. This test case has a simple non-replicated clustered object. Unlike Case 4, we use the full SCOPE dereferencing mechanism. This dereferencing consists of deciding which VP the dereference takes place on, finding that VP's table, calculating an offset into that table, getting a new address from that table location and going to that address. We expect this to be the slowest of all the counters, as it has the sharing effects and the overhead from the SCOPE dereference.

Case 6: is one ReplicatedIntegerCounter clustered object. This test case has a replicated clustered object. In this case there is one replica per processor. The *int* field that we are incrementing is in these replicas. When an `inc()` operation takes place the *int* on the local replica is incremented. When a value operation is requested the ReplicatedIntegerCounter's root provides a linked list of all the replicas that we can traverse to find the summation of each processor's *int* values. We expect the ReplicatedIntegerCounter to perform very well as there is no sharing.

Average cycles per inc call and percent overhead of sharing		
Case 1: One IntegerCounter	681.29	86.7%
Case 2: ArrayIntegerCounter	548.05	79.1%
Case 3: PaddedArrayIntegerCounter	92.10	3.8%
Case 4: CO style IntegerCounter	749.50	90.8%
Case 5: SharedIntegerCounter	885.44	88.8%
Case 6: ReplicatedIntegerCounter	100.11	$< 1.0\%$

Table 5.2. *The results from the 6 test cases mentioned in Section 5.2.3*

The experiment is the same as that of Section 5.1.7: the IntegerCounters are subjected to a number of concurrent `inc()` calls and the time taken to service these calls is measured. An average is taken across several runs to give us the average number of cycles per request. We also add one special run of the test on only one thread to find out the underlying cost of the counters regardless of the effects of sharing.

5.2.4 Results

Table 5.2 presents the results obtained for the second experiment, and Figure 5.5 summarizes the numbers from Table 5.2 in a graphical form.

5.2.5 Analysis

The results from the first three test cases are (as expected) similar to the first experiment. The sharing and false sharing in Case 1 and 2 causes large slowdowns. Case 3 mitigates the sharing and hence does not experience the same slowdowns. Case 4 exhibits the sharing seen in Case 1 and 2, but still is 15% faster than a clustered object using the dereference mechanism; however, Case 4 is 10% slower than Case 1 where we used the simpler class. Case 5 exhibits the same sharing as Case 1, 2 and 3, but also on top of the sharing Case 5 is slowed further by the more complex dereference mechanism of SCOPE. Case 6 exhibits no sharing, as expected for a fully replicated clustered object. The performance is only

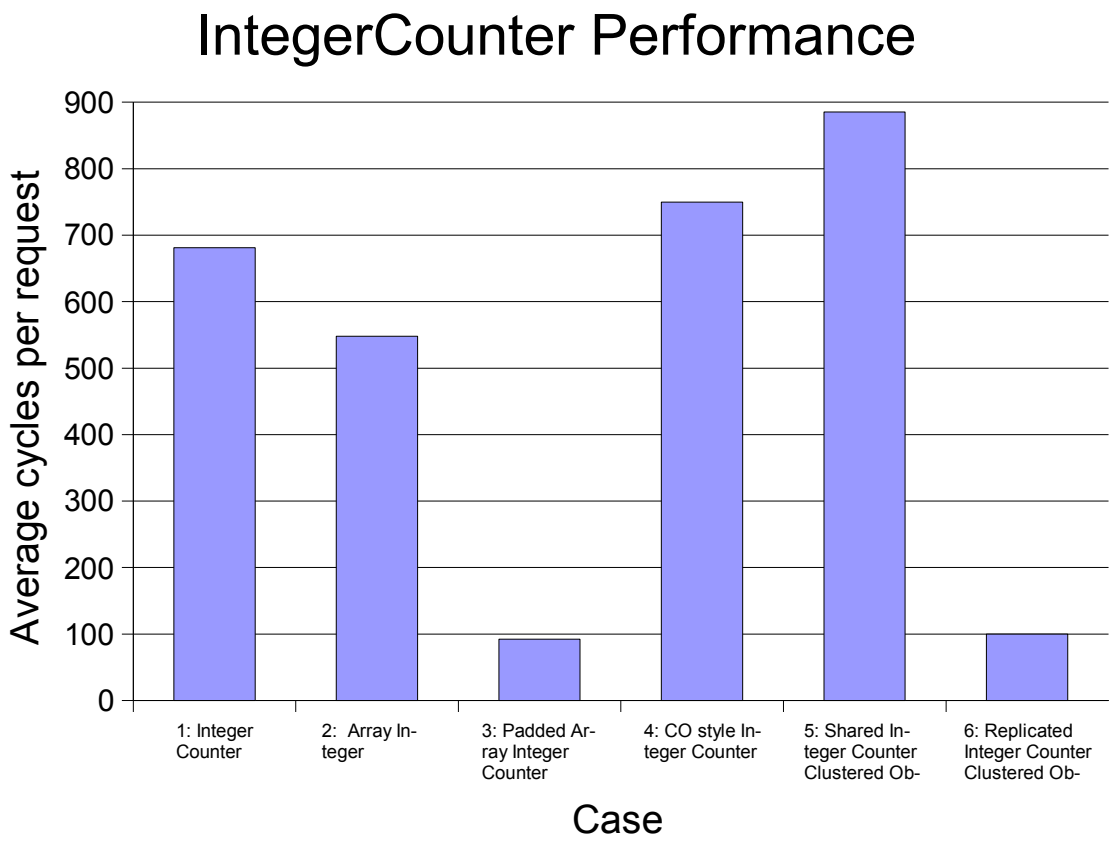

Figure 5.5. *Average runtime results of the six IntegerCounters listed in Section 5.2.3, Table 5.2.*

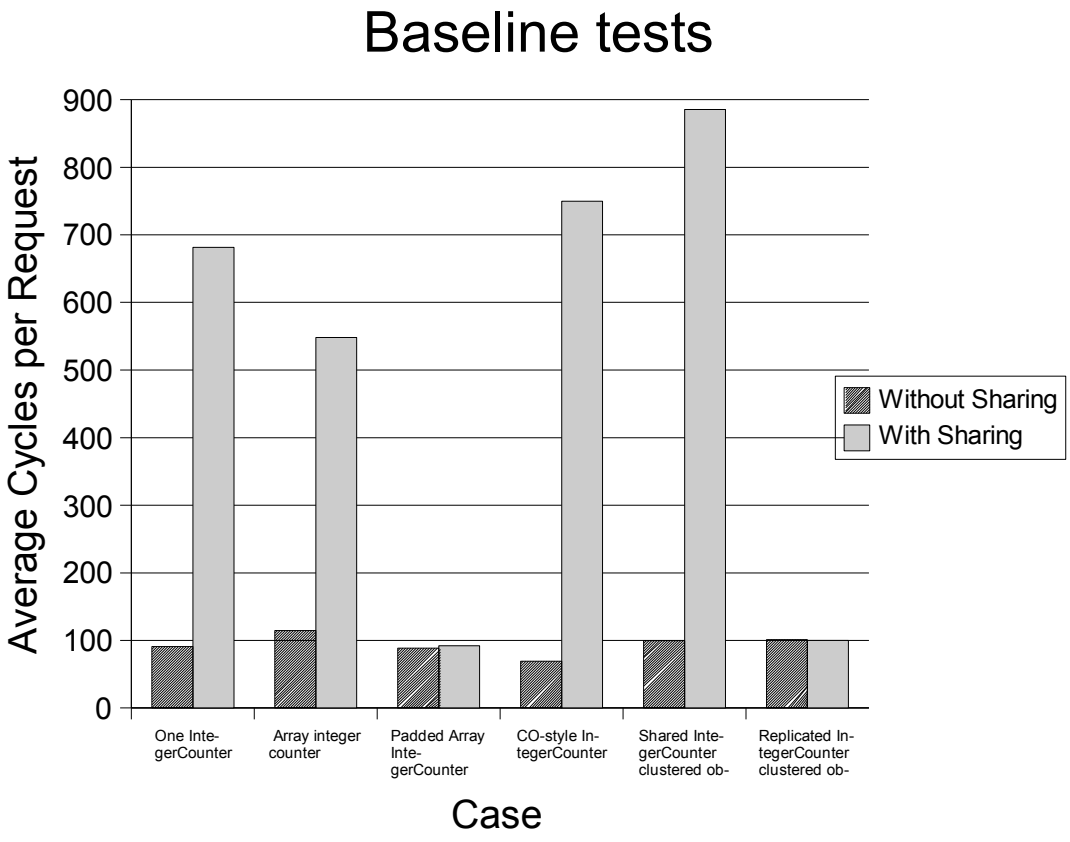

Figure 5.6. *Average runtime results of the six IntegerCounters listed in Section 5.2.3 running on one processor. This effectively eliminates sharing.*

slightly worse (9%) than that of the PaddedArrayIntegerCounter. The difference is due to the different dereference system and differences in the classes.

The second set of tests shown in Figure 5.6 reveals only a small difference between each of the methods when there was no sharing present. The difference between the clustered object style counter and replicated clustered object is relatively small, 20 cycles. We can attribute this difference to our SCOPE lookup mechanism.

In terms of performance, we believe this shows that a user-level library can indeed provide a performance increase by maximizing locality in this simple counter example.

5.3 Reproduction of Benefits

Section 2.3 lists the benefits of using clustered objects in a system. Table 2.1 also provides a summary of these benefits.

From a developer's perspective, clustered objects are supposed to reduce the use of ad hoc mechanisms for increasing locality. Earlier in this chapter we introduced a PaddedArrayIntegerCounter. This counter is an excellent example of a data structure that is structured correctly to reduce sharing, but was done so in an ad hoc manner. The ReplicatedIntegerCounter that was also introduced earlier is one possible implementation of the data structure as a clustered object. Figure 5.7 shows the PaddedArrayIntegerCounter's code, and Figure 5.8 shows the clustered object equivalent. There are several ad hoc mechanisms used in the PaddedArrayIntegerCounter. On lines 12 and 14 of Figure 5.7, when the array is allocated, the entries have to be padded to avoid false sharing. On line 19 of Figure 5.7 the base pointer has to be multiplied by the number of the current processor and the cache line size, and a similar multiplication has to happen to increment or decrement on line 23.

In the ReplicatedIntegerCounter from SCOPE in Figure 5.8, there one integer per rep, and it is accessed just as a regular integer would be. None of those ad hoc mechanisms listed above are needed. When creating the integer storage, no special treatment is necessary, the

```
1  class PaddedArrayIntegerCounter: public IntegerCounter {
2  private:
3      volatile int* _vals;
4  public:
5      PaddedArrayIntegerCounter();
6      virtual void value(int &count,int MYVP);
7      virtual void inc(int MYVP);
8      virtual void dec(int MYVP);
9  };
10
11 PaddedArrayIntegerCounter::PaddedArrayIntegerCounter(){
12     malloc(CACHE_LINE_SIZE);
13     _vals = (volatile int*)
14     malloc(CACHE_LINE_SIZE*NUM_OF_PROCESSORS);
15 }
16
17 void PaddedArrayIntegerCounter::value(int &count,int MYVP){
18     for(int i = 0; i < NUM_OF_PROCESSORS; i++)
19      count += (int) _vals+(i*CACHE_LINE_SIZE);
20 }
21
22 void PaddedArrayIntegerCounter ::inc(int MYVP){
23     FetchAndAddintSynced(_vals+(MYVP*CACHE_LINE_SIZE),1);}
24
25 void PaddedArrayIntegerCounter ::dec(int MYVP){
26     FetchAndAddintSynced(_vals+(MYVP*CACHE_LINE_SIZE),-1);}
```

Figure 5.7. *The PaddedArrayIntegerCounter implementation. In this example MYVP retrieves the number of the current processor.*

```
1  class ReplicatedIntegerCounter : public IntegerCounter {
2      volatile int _val;
3      ReplicatedIntegerCounter();
4      ~ReplicatedIntegerCounter() { }
5  public:
6      static CORC Create(IntegerCounterRef &ref);
7      virtual CORC value(sval &count);
8      virtual CORC inc();
9      virtual CORC dec();
10 };
11 ReplicatedIntegerCounter::ReplicatedIntegerCounter()
12                     :IntegerCounter(),_val(0) {}
13 CORC ReplicatedIntegerCounter::Create(IntegerCounterRef &ref) {
14     CORef r;
15     Root::Create(r);
16     ref = (IntegerCounterRef)r;
17     return SUCCESS;
18 }
19 CORC ReplicatedIntegerCounter::value(sval &count) {
20     ReplicatedIntegerCounter* current = NULL;
21     count = 0;
22     for (void* curr = COGLOBAL(nextRep(NULL,(CObjRep*&)current));
23          curr;curr = COGLOBAL(nextRep(curr,(CObjRep*&)current))) {
24          count+=current->_val;}
25     return SUCCESS;
26 }
27 CORC ReplicatedIntegerCounter ::inc(){
28     FetchAndAddintSynced(&_val,1);
29     return SUCCESS;
30 }
31 CORC ReplicatedIntegerCounter ::dec() {
32     FetchAndAddSvalSynced(&_val,-1);
33     return SUCCESS;
34 }
```

Figure 5.8. *The ReplicatedIntegerCounter implementation. In this example the MYVP macro retrieves the number of the current processor. The locking in this example has been omitted for simplicity.*

padding is automatic. When iterating across the integers to get the total value, a simple for each rep loop is used. When incrementing or decrementing, the local value is used.

In a data structure as simple as a counter, the complexity introduced by these ad hoc mechanisms might be bearable, but as the complexity of the data structure increases, so does the complexity of the locality management. Just as in previous kernel-level implementations, SCOPE removes the need for ad hoc mechanisms that provide locality; but, how does SCOPE measure up to the K42 implementation of clustered objects? The following sections evaluate the original benefits listed in Section 2.3, and evaluate our implementation on the basis of this list.

5.3.1 Programming Benefits

An evaluation of the benefits to the programmer:

Ease of use encouraged by:

1. Reduction in the use of ad hoc mechanisms for increasing locality: This is inherent in the model, and therefore in SCOPE. Our simple integer counter above shows a fully distributed version that is simple to understand and use.
2. For the programmer, accessing a clustered object is no harder than a regular object access: We have changed the clustered object access mechanisms. As explained in Section 4.2.2 we have added two macros. We believe this is no harder for the programmer than using the original DREF macro. In Section 6.1.3 we describe mechanisms that might allows us to make this system even easier.
3. Assisted destruction is not implemented, however the in Section 6.1.1.2 explains how we plan to do this.
4. Incremental optimizations: This is inherent in the clustered object model and therefore in SCOPE.
5. No existence locks: is a feature of assisted destruction.

Linguistic features supported:

1. Strong interfaces: This is inherent in the clustered object model and therefore in SCOPE.
2. Type safety: The new DREF macro is type safe, therefore SCOPE maintains type safety.
3. Oblivious structure: This is inherent in the clustered object model and therefore in SCOPE.

Most of the programming benefits of clustered objects have been maintained in the implementation of SCOPE, with plans to implement the rest.

5.3.2 Utilization

Besides the overall performance analyzed earlier in this chapter, SCOPE's support of utilization specific features:

enables utilization by:

1. Structure facilitates reduced sharing: This is inherent in the clustered object model and therefore in SCOPE.
2. Dynamic adaptation: is not yet implemented. Section 6.1.1.4 of the future work explains how we would like to implement this.
3. Different implementations: This is inherent in the model, and therefore in SCOPE.

aids efficiency by:

1. Minimal overhead: Originally, dereferencing a clustered object took 1 extra instruction. We show above in Section 5.2.4 that the SCOPE dereference mechanism takes approximately 20 cycles. This does introduce more overhead, but we also show that even with this overhead, SCOPE is much faster because of the increased locality. Further work could be applied to creating a faster DREF mechanism.

Benefit		status in SCOPE
Programming		
ease of use	reduce ad hoc mechanisms	supported in SCOPE
	easy access	
	easy destruction	not implemented
	no existence locks	
language features	promotes optimization	supported in SCOPE
	strong interface	
	type safe	
	oblivious clients	
Utilization		
enables utilization	dynamic adaptation	not implemented
	multiple implementation	
	easy incremental optimization	supported in SCOPE
increases efficiency	no existence locks	not implemented
	minimal access overhead	not as good as K42
	lazy creation of reps	supported in SCOPE

Table 5.3. *Summary of the Benefits provided by SCOPE*

2. Lazy creation: is supported by the trampoline mechanism in SCOPE.

Most of the utilization features of clustered objects have been maintained in the implementation of SCOPE, with plans to implement the rest. A summary of each of the benefits is listed in Table 5.3.

5.4 Summary

We have shown that sharing can be a problem for user-level programs, and that a simple restructuring of the data can avoid the sharing problem. We have also shown that in a simple example SCOPE can remove the effects of sharing, just as in a manually structured object. Finally we compare the benefits established by previous clustered object systems, and show that SCOPE has accomplished most of them, and those that were not, could be implemented in future versions of the prototype.

Chapter 6

Future Work and Conclusions

This chapter concludes the thesis by discussing work up to this point, lessons learned, and work that could be done in the future.

SCOPE has provided a fruitful platform to continue research. Further work can happen in two areas: furthering the development of SCOPE and furthering the implementation strategies of clustered objects themselves.

6.1 Future Work

This section features two areas of possible future research. First, we consider some advanced features for SCOPE then we consider some improvements to the current implementation of SCOPE. Finally, we describe how one might be able to use aspect-oriented programming to enhance the clustered objects user's experience.

6.1.1 Reproduction of Advanced Features

K42's Clustered Object runtime is relatively mature, and has had time to evolve some advanced features. In the interest of the SCOPE implementation time line, these features were not implemented, but could be in the future. These features include things like clustered objects' dynamic monitoring system, garbage collection, RCU, and dynamic update systems. Beyond K42's features, the portability of SCOPE could be demonstrated with a Microsoft Windows version.

6.1.1.1 KORE

The K42 Objects Remote Environment (*KORE*), is a system designed to provide dynamic information about clustered objects in K42. KORE loads a server into the clustered object's runtime environment, then a KORE client or GDB[1] client is able to connect to that server and collect information about the system.

KORE can do things like list clustered objects in the system, and monitor events and make traces. KORE's client is scriptable, so users can write their own scripts to mine runtime information from the system.

The KORE client should not need to be changed to be work with SCOPE, and the most of the original KORE server implementation should work with. Interesting questions include whether we can reduce KORE's overhead by reducing its communication overhead.

6.1.1.2 Garbage Collection

K42's Clustered Objects has what they call garbage collection. This is not garbage collection in the traditional sense of automatic reclamation of unused objects, but rather a semi automatic cleanup of objects that have been signaled for destruction.

K42's garbage collection is what allows them to make the existence grantees on clustered objects. Any reference to a clustered object is guaranteed to point to an active clustered object. Even after the destruction of a clustered object is signaled, it is kept active until all threads in the system are no longer able to access it. It does this in the same way that the RCU technique mentioned is Section 1.3.3 makes its guarantees about writing data with quiescent periods: it waits until all threads in the system are no longer able to use their current reference. In K42 they have a policy of having no *long living* threads. This means that all threads should terminate in a reasonable amount of time. Since OSes are request driven, this plays out as a single short lived thread per-request. So, after a clustered object's destruction is signaled, the system just waits until all the requests that were active

[1] GDB is a commonly used open source debugger: http://www.gnu.org/software/gdb/

at the time finish, then destroys the clustered object. Because clustered objects are always guaranteed to exist, they have the desirable property of not needing existence locking. This is considered one of the nicer benefits of clustered objects.

The portability events we introduce in Section 4.2.2 will make implementing garbage collection simple; however, whether the expectation of user-level programs not having any long living threads is reasonable is yet to be seen.

6.1.1.3 RCU

The RCU technique mention in Section 1.3.3 is used pervasively in K42. Clustered objects tries to control sharing, but it may be possible that a programmer needs a shared piece of data in a clustered object. RCU could be used as a lock-free method of accessing data that must be shared. It seems reasonable for SCOPE to provide this functionality. It seems like RCU's quiescent state detection requires OS integration, it will prove to be an interesting challenge to move this methodology into user-level. Since the garbage collection mentioned above uses a RCU style technique, it may be able to be extended to provide full RCU support.

6.1.1.4 Dynamic Update and Hot Swapping

K42's Clustered Objects support *dynamic update*, replacing all of a type of a clustered object with a different version of the same type clustered object at runtime. K42's *hot swapping* allows one version of a single clustered object to be replaced with another version at runtime. To support this, K42 has a set of factory objects for creating and tracking clustered objects. Once again, the portability events introduced in Section 4.2.2 will allow SCOPE to easily accommodate dynamic updates.

6.1.1.5 Portability

In Section 4.2.3 we talk about how the portability events will allow SCOPE to work in other systems. It could prove interesting to try moving SCOPE to the Microsoft Windows

platform, and then revalidate the performance and portability of SCOPE.

6.1.2 SCOPE Improvements

After becoming well acquainted with SCOPE's internal structure, we found several flaws that could be fixed in a simple redesign. At the moment the lookup tables and the COSMgr are very tightly integrated. There is no need for this. The method for accessing external resources on different platforms is ad hoc, and does not seem to scale as more interfaces to the underlying system are added. The internal naming conventions are not always observed. There is no documentation about the system.

SCOPE may also have a dependence on GCC 3.4[2]. GCC 3.4 is used exclusively in K42. For widespread adoption, it would be necessary to support newer versions of GCC and even other compilers.

6.1.3 Improving the Client experience with AOP

The use of Clustered Objects in K42 is not completely transparent for the programmers of K42. The *DREF* macro must be used to access a clustered object, so the programmer must use this macro every time a call to a clustered object is made. In the systems domain, this is considered a good calling convention because the extra overhead caused by the *DREF* macro is made explicit. The virtual dispatch used in object oriented programming has a similar per call overhead; however, its use is transparent to the caller. The transparent virtual dispatch has been accepted in many user-level applications, so it seems reasonable that if there were a transparent clustered object dereference mechanism, it may be accepted at user-level as well.

Aspect Oriented Programming (*AOP*) provides mechanisms for concretely implementing cross-cutting concerns in a modular fashion. For our purposes a cross-cutting concern is a feature that is not easily implemented in one module in the system. One example of a

[2]GCC is a popular open source compiler: http://gcc.gnu.org/

cross-cutting concern in SCOPE could be the clustered object creation code, when creating a clustered object, code from all around the system is called in a predetermined way.

One of the controversial properties of AOP that may be useful to us is *obliviousness*. That is, that when an aspect acts on some part of the system, the original code does not look any different. One drawback of clustered objects is that without compiler support, a clustered object access has to be surrounded in the *DREF* macro as mentioned in Section 4.2.1. This means that the programmer has to be aware that it is a clustered object that is being called, not a regular object. AOP could be used to make this more transparent by applying the *DREF* macro automatically to clustered object calls, therefore making them look like regular object accesses to the programmer. Making clustered objects even easier to call could help them become more pervasive in programs, which is one of the original goal of clustered objects and this work.

Automatically applying the *DREF* macro may have even more advantages, in that the added *DREF* calls could be customized for the calling clustered object. If for example, there was a long lived clustered object like the COSMgr that will never need to be garbage collected, or if a clustered object that didn't use a RCU facility, the necessary tracking information could not be collected. This may have the effect of creating faster access times for clustered objects that opt out of advanced features.

One problem with this (and AOP in general) is that the client program would have to first be compiled with an AOP compiler before being regularly compiled; however, it may be possible to prepackage an AOP compiler that is setup to just process the clustered object aspects like a simple preprocessor.

6.2 Conclusions

The evolution of OSes to modern day systems with SMP support has had to change how concurrency is dealt with in order to provide better CPU utilization. One system that evolved from the need for better SMP utilization was K42's Clustered Objects facility.

Clustered Objects' primary goal was to help the programmer maximize locality in a structured manner. The clustered objects model offer improved scalability with respect to the problems faced by modern OSes, but the problems caused by sharing do not just affect OSes, they affect all software.

SCOPE was created to bring the K42 implementation of the clustered objects model into the Linux user-level. Porting the K42 Clustered Objects facility is not a simple task, there were many dependencies on the underlying OS, including the object translation system, the IPC system, and the memory allocation system.

To help increase the portability of SCOPE we introduce the notion of a portability event. Portability events allow us to customize the user-level runtime environment to be able to provide the object translation support that clustered objects needs to function. With the addition of the portability event, we can replace the K42 dereference functionality with our own which allows us to produce a processor specific table lookup. This macro has a higher overhead than the original, but produces the same result, a clustered object dereference.

The implementation of the support for the new portability events was successful, in that we were able to produce a clustered object lookup at runtime, and reproduce much of the functionally of the Clustered Objects facility in K42.

In our evaluation of SCOPE, we first showed that sharing can be a problem for user-level programs by introducing a simple integer counter. We established that without some extra mechanisms beyond what a programmer might regularly do, a simple counter can experience significant sharing overheads even on a dual processor machine; however, a systematic manual restructuring of the data can avoid the sharing problem and increase utilization of the system's resources like the CPUs.

After we established sharing can be a problem for our test situation, we then show that in a simple example SCOPE can remove the effects of sharing, just as in a manually restructured structured object. Then we compare the performance of SCOPE to the manual restructuring, and find that the overhead introduced by SCOPE is small, especially relative to the slowdowns experienced from the effects of sharing. Finally, we compare the benefits

established by previous implementations of clustered object systems, and show that SCOPE has most of them, and that the missing benefits could be implemented in later versions of SCOPE.

The question addressed in this work is whether, in the face of the system level complexities introduced by true concurrency, the use of an abstraction at user-level could be shown to increase utilization without customization of an underlying system? We believe we have demonstrated the answer is yes.

The results of this thesis show that even though SCOPE does not rely on the underlying OS for anything besides its basic services, SCOPE is able to help a programmer write truly concurrent software by easing the need to dwell on the lowest details of the system and by systematically maximizing hardware utilization.

Bibliography

[1] AMD, "Welcome to AMD multi-core technology," 2005. [Online]. Available: http://multicore.amd.com/Global/

[2] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," in *SIGMETRICS '89: Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1989, pp. 49–60.

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

[4] ——, "Clustered Objects," Ph.D. dissertation, University of Toronto, 2005.

[5] ——, "Personal communications with Jonathan Appavoo," September-December 2005.

[6] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, "K42 overview," IBM TJ Watson Research, Tech. Rep., 2002.

[7] G. Barnes, "A method for implementing lock-free shared-data structures," in *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM Press, 1993, pp. 261–270.

[8] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*. New York, NY, USA: ACM Press, 2000, pp. 195–203.

[9] I. Bowman, S. Siddiqi, and M. Tanuan, "Concrete architecture of the linux kernel," 1998. [Online]. Available: http://plg.uwaterloo.ca/~itbowman/CS746G/a2/

[10] S. Bray, M. Yuen, Y. Coady, and M. E. Fiuczynski, "Managing variability in systems: Oh what a tangled OS we weave," in *Proceedings of the MVCDC Workshop, OOPSLA*, Vancouver, British Columbia, Canada, 2004.

[11] R. Bryant, J. Hawkes, and J. Steiner, "Scaling Linux to the extreme: from 64 to 512 processors," in *Ottawa Linux Symposium*, 2004.

[12] M. C. Daniel P. Bovet, *Understanding the Linux Kernel, Second Edition*, A. Oram, Ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[13] E. W. Dijkstra, "Cooperating sequential processes," Technological University Eindhoven, Tech. Rep., 1965.

[14] ——, "The structure of "THE"-multiprogramming system," *Comm.ACM*, vol. 11, no. 5, pp. 341–346, 1968.

[15] M. E. Fiuczynski, "Better tools for kernel evolution, please!" *;LOGIN:*, vol. 30, no. 5, pp. 8–10, 2005.

[16] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker, "patch (1) considered harmful," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*. Santa Fe, New Mexico, USA: IEEE Computer Society, 2005.

[17] S. L. Gaede, "Perspectives on the SPEC SDET benchmarks," January 1999. [Online]. Available: http://www.spec.org/osg/sdm91/sdet/index.html

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," *Lecture Notes in Computer Science*, vol. 707, pp. 406–431, 1993. [Online]. Available: citeseer.ist.psu.edu/gamma93design.html

[19] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 1999, pp. 87–100.

[20] T. Harris and K. Fraser, "Language support for lightweight transactions," in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. New York, NY, USA: ACM Press, 2003, pp. 388–402.

[21] B. Henderson, "Linux loadable kernel module howto," 2006. [Online]. Available: http://tldp.org/HOWTO/Module-HOWTO/

[22] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[23] IEEE, *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, 1995. [Online]. Available: http://www.ansi.org/

[24] O. Krieger, M. Stumm, R. Unrau, and J. Hanna, "A fair fast scalable reader-writer lock," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II - Software. Boca Raton, FL: CRC Press, 1993, pp. II–201–II–204. [Online]. Available: citeseer.ist.psu.edu/krieger93fair.html

[25] P. E. McKenney, J. Slingwine, and P. Krueger, "Experience with an efficient parallel kernel memory allocator," *Softw. Pract. Exper.*, vol. 31, no. 3, pp. 235–257, 2001.

[26] P. E. McKenney and J. D. Slingwine, "Read-Copy Update: Using execution history to solve concurrency problems," in *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998, pp. 509–518.

[27] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[28] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm, "(de-)clustering objects for multiprocessor system software," in *Fourth International Workshop on Object Orientation in Operating Systems 95 (IWOOO'95)*, 1995, pp. 72–81.

[29] M. Shapiro, "Structure and encapsulation in distributed systems: the proxy principle," in *Proceedings of the 6th Int. Conf. on Distributed Systems (ICDCS)*, Cambridge MA (USA), May 1986, pp. 198–204.

[30] M. Shapiro, Y. Gourbant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot, "SOS: An object-oriented operating system - assessment and perspectives," *Computing Systems*, vol. 2, no. 4, pp. 287–337, 1989.

[31] N. Shavit and D. Touitou, "Software Transactional Memory," in *Symposium on Principles of Distributed Computing*, 1995, pp. 204–213. [Online]. Available: citeseer.ist.psu.edu/shavit95software.html

[32] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[33] B. Smith, "The quest for general-purpose parallel computing," in *Developing a computer science agenda for high-performance computing*. New York, NY, USA: ACM Press, 1994, pp. 151–155.

[34] W. Stallings, *Operating Systems Internals and Design Principles Third Edition*. Prentice Hall, 1997.

[35] D. A. Wheeler, "Program library howto," 2003. [Online]. Available: http://www.tldp.org/HOWTO/Program-Library-HOWTO/

[36] G. Yilmaz and N. Erdogan, "Partitioned object models for distributed abstractions," in

14th International Symp. on Computer and Information Sciences (ISCIS XIV). Kusadasi, Turkey: IOS Press, 1999, pp. 1072–1074.

Appendix A

Test Machines

A.1 Dual Processor X86

This section has the output produced by the CPU-Z[1] program. Table A.1 has the detailed CPU information, Table A.2 the motherboard information, and Table A.3 the memory configuration.

[1] http://www.cpuid.com/cpuz.php

	CPU#1	CPU #2
Name	AMD Athlon MP	AMD Athlon MP
Code Name	Thoroughbred	Thoroughbred
Specification	AMD Athlon(tm) MP 2400+	AMD Athlon(tm) MP 2400+
Family / Model / Stepping	6 8 1	6 8 1
Extended Family / Model	7 8	7 8
Package	Socket A	Socket A
Core Stepping	B0	B0
Technology	0.13 um	0.13 um
Supported Instructions Sets	MMX, Extended MMX, 3DNow!, Extended 3DNow!, SSE	MMX, Extended MMX, 3DNow!, Extended 3DNow!, SSE
CPU Clock Speed	2000.1 MHz	2000.1 MHz
Clock multiplier	x 15.0	x 15.0
Front Side Bus Frequency	133.3 MHz	133.3 MHz
Bus Speed	266.7 MHz	266.7 MHz
L1 Data Cache	64 KBytes, 2-way set associative, 64 Bytes line size	64 KBytes, 2-way set associative, 64 Bytes line size
L1 Instruction Cache	64 KBytes, 2-way set associative, 64 Bytes line size	64 KBytes, 2-way set associative, 64 Bytes line size
L2 Cache	256 KBytes, 16-way set associative, 64 Bytes line size	256 KBytes, 16-way set associative, 64 Bytes line size
L2 Speed	2000.1 MHz (Full)	2000.1 MHz (Full)
L2 Location	On Chip	On Chip
L2 Data Prefetch Logic	yes	yes
L2 Bus Width	64 bits	64 bits

Table A.1. *CPU information for the test machine*

Motherboard manufacturer	Gigabyte Technology Co., LTD
Motherboard model	7DPXDW-P, x.x
BIOS vendor	Award Software International, Inc.
BIOS revision	6.00 PG
BIOS release date	04/25/2003
Chipset	AMD AMD-762 rev. A0
Southbridge	AMD AMD-768 rev. 05
Sensor chip	Winbond W83627HF
Graphic Interface	AGP
AGP Status	enabled, rev. 2.0
AGP Data Transfer Rate	4x
AGP Max Rate	4x
AGP Side Band Addressing	supported, enabled

Table A.2. *Mainboard and chipset of the test machine*

DRAM Type	DDR-SDRAM
DRAM Size	1024 MBytes
DRAM Frequency	133.3 MHz
FSB:DRAM	1:1
CAS# Latency	2.0 clocks
RAS# to CAS#	3 clocks
RAS# Precharge	3 clocks
Cycle Time (TRAS)	6 clocks
Bank Cycle Time (TRC)	9 clocks
DRAM Idle Timer	8 clocks
# of memory modules	2
Module 0	DDR PC2700 - 512 MBytes
Module 1	DDR PC2700 - 512 MBytes

Table A.3. *Memory of the test machine*

Appendix B

Integer Counters

B.1 The Integer Counter Interface

```
class IntegerCounter{
public:
    virtual void inc(int MYVP)              =0;
    virtual int dec(int MYVP)               =0;
    virtual int value(int &count,int MYVP) =0;
};
```

These are the methods that all of our integer counters had. *MYVP* stands for my virtual processor. Virtual processors are an abstraction used in K42 to represent processors. MYVP is just an integer that is the number of the processor: 0 or 1 in the case of our first test.

B.2 Atomically Accessing an Int

```
1  inline void FetchAndAddintSynced(volatile int* datap, int val)
2  {
3      int oldData;
4      int rcode;
5
6      while (1) {
7      __asm__ __volatile__ (
8              "#fetchAndAdd
9              movl (%3), %1
10             add %1, %2
11             lock
12             cmpxchg %2, (%3)
```

```
13              jz 1f
14              movl $0, %0
15              jmp 2f
16              1:  movl $1, %0
17              2:  # end fetchAndAdd"
18      : "=r" (rcode), "=&a" (oldData)
19      : "r" (val), "r" (datap)
20      : "memory");
21      if (rcode) return; /*oldData;*/
22      }
23  }
```

B.3 The Simple Integer Counter

```
1   class SimpleIntegerCounter: public IntegerCounter
2   {
3   private:
4       volatile int* _val;
5   public:
6       SimpleIntegerCounter();
7
8       virtual void value(int &count,int MYVP);
9
10      virtual void inc(int MYVP);
11
12      virtual void dec(int* MYVP);
13
14  };
15
16  SimpleIntegerCounter ::SimpleIntegerCounter(){
17      _val = (volatile int*) malloc(sizeof(int));
18  }
19
20  void
21  SimpleIntegerCounter::value(int &count,int MYVP){
22       count = *_val;
23  }
24
25  inline void
26  SimpleIntegerCounter ::inc(int MYVP){
27      FetchAndAddintSynced(_val,1);
```

```
28 }
29
30 inline void
31 SimpleIntegerCounter ::dec(int MYVP){
32     FetchAndAddintSynced(_val,-1);
33 }
```

B.4 Array Based Integer Counter

```
1  class SimpleArrayIntegerCounter: public IntegerCounter
2  {
3  private:
4      volatile int* _vals;
5  public:
6      SimpleArrayIntegerCounter();
7
8      virtual void value(int &count,int MYVP);
9
10     virtual void inc(int MYVP);
11
12     virtual void dec(int MYVP);
13 };
14
15 SimpleArrayIntegerCounter::SimpleArrayIntegerCounter(){
16     malloc(CACHE_LINE_SIZE);//add a pad
17     _vals = (volatile int*)
18                 malloc(sizeof(int)*NUM_OF_PROCESSORS);
19 }
20
21 void
22 SimpleArrayIntegerCounter::value(int &count,int MYVP){
23     for(int i = 0; i < NUM_OF_PROCESSORS; i++)
24         count += _vals[i];
25 }
26
27 inline void
28 SimpleArrayIntegerCounter ::inc(int MYVP){
29     FetchAndAddintSynced((_vals+MYVP),1);
30 }
31
32 inline void
```

```
33 SimpleArrayIntegerCounter ::dec(int MYVP){
34     FetchAndAddintSynced((_vals+MYVP),-1);
35 }
```

B.5 Padded Array Based Integer Counter

```
1  class SimplePaddedArrayIntegerCounter: public IntegerCounter
2  {
3  private:
4      volatile int* _vals;
5  public:
6      SimplePaddedArrayIntegerCounter();
7
8      virtual void value(int &count,int MYVP);
9
10     virtual void inc(int MYVP);
11
12     virtual void dec(int MYVP);
13
14
15 };
16 SimplePaddedArrayIntegerCounter::SimplePaddedArrayIntegerCounter(){
17     malloc(CACHE_LINE_SIZE);
18     _vals = (volatile int*)
19         malloc(CACHE_LINE_SIZE*NUM_OF_PROCESSORS);
20
21 }
22
23 void
24 SimplePaddedArrayIntegerCounter::value(int &count,int MYVP){
25     for(int i = 0; i < NUM_OF_PROCESSORS; i++)
26      count += (int) _vals+(i*CACHE_LINE_SIZE);
27 }
28
29 inline void
30 SimplePaddedArrayIntegerCounter ::inc(int MYVP){
31     FetchAndAddintSynced(_vals+MYVP,1);
32 }
33
34 inline void
35 SimplePaddedArrayIntegerCounter ::dec(int MYVP){
```

```
36      FetchAndAddintSynced(_vals+MYVP,-1);
37  }
```

In this particular example we set MYVP to be the processor number times the size of a cache line to save the multiplication each time.

Appendix C

Single User Mode

A common practice in testing operating system components and other things that are very sensitive to changes in the system is to run the test in Linux's single user mode. Single user mode allows a program to run on a very minimal system, where only the processes that are necessary to maintain the system are running.

In our tests we want two processes to be running at the same time, while accessing the same data. Any time that one processor spends servicing non-test processes throws off the results because there is no longer contention for the data from both processors. It is advantageous for us to minimize the amount of other processes able to interfere with the test.

To tell whether single user mode testing was necessary, we ran the first test, in both single user mode (level 1), and regular (level 5) mode. Figure C.1 shows the data set obtained from both. Notice that single user mode results are much more regular.

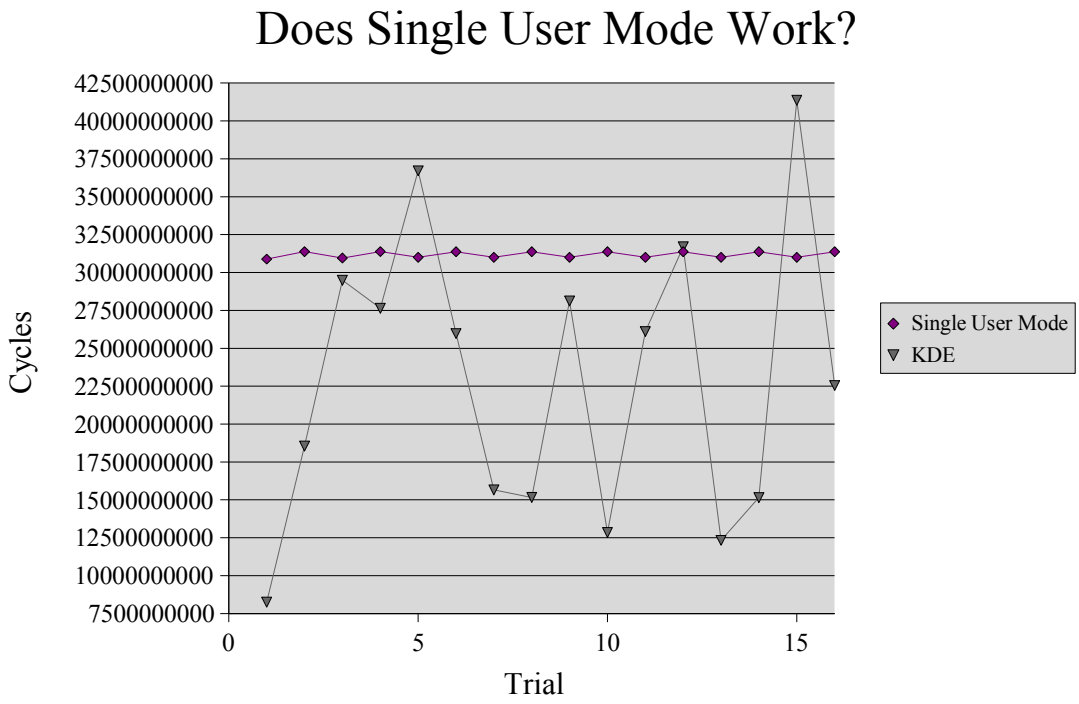

Figure C.1. *Running a simple test in single user mode, and regular mode. Single user mode provides more regular results.*

Title of Thesis:

SCOPE: Scalable Clustered Objects with Portable Events

Author: _____

CHRISTOPHER JAMES MATTHEWS

August 9, 2006

THESIS WITHHOLDING FORM

At our request, the commencement of the period for which the partial licence shall operate shall be delayed from August 9, 2006 for a period of at least six months.

_____

(Supervisor)

_____

(Department Chairman)

_____

(Dean of Graduate Studies)

_____

(Signature of Author)

_____

(Date)

Date Submitted to the Dean's Office: _____