Clustered Objects: Initial Design, Implementation and Evaluation

by

Jonathan Appavoo, B.Sc.

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

Clustered Objects: Initial Design, Implementation and Evaluation

Jonathan Appavoo, B.Sc.
University of Toronto, 1998

Supervisor: Michael Stumm

To achieve high performance on shared memory multiprocessors, software must be designed to take locality into account. The appropriate use of replication, partitioning and sharing of data can lead to higher locality, thus reducing communication and synchronization overheads. *Clustered Objects* is a partitioned object model exclusively targeted at expressing locality optimizations in a consistent manner. Like other partitioned object models, Clustered Objects allow an object to be decomposed into multiple representative objects while preserving a single unified external interface.

In this dissertation we develop and implement a model for Clustered Objects based on support found in the Tornado Operating System. Although preliminary, our experimental work indicates that developing software with this model benefits from the software engineering advantages of object-oriented programming and yet is structured to exploit fine-grained locality optimizations.

Acknowledgments

My deepest thanks and gratitude go to Professor Stumm for all his help, advice, and support. His patience and judgment as a supervisor have made this thesis possible. I would also like to thank Benjamin Gamsa. Clustered Objects along with Tornado would not exist without Ben's creativity, insight and ability. I am deeply indebted to him for suffering all my questions.

Finally my wife, Ariane Appavoo deserves considerable credit for this thesis and all my successes. Her love and support was not only expressed in her never-ending belief in me but also in her tolerance of my less than pleasant moods. The drafts of this thesis were only intelligible due to her patient readings.

Jonathan Appavoo

University of Toronto

October 1998

Contents

Chapter 1

Introduction

The development of high-performance parallel systems software is a difficult task. The concurrency and locality management needed for good performance can add considerable complexity. Clustered Objects were developed as a model of *partitioned objects* to simplify the task of designing high-performance shared memory multiprocessor (SMP) systems software. In the partitioned object model, an externally visible object is composed of a set of distributed representative objects. Each representative object locally services requests, possibly collaborating with one or more other representatives. Cooperatively all the representatives implement the complete functionality of the Clustered Object. The distributed nature of Clustered Objects make them ideally suited for the design of multiprocessor system software, which often requires a high degree of modularity and yet benefits from the sharing, replicating and partitioning of data on a per-resource (object) basis. Clustered Objects are similar to other partitioned object models, such as Fragmented Objects [22, 5] and Distributed Shared Objects[38, 14], although the latter have focused on the requirements of (loosely coupled) distributed environments; Clustered Objects are designed for (tightly coupled) shared memory systems.

The work presented in this dissertation is an initial evaluation of Clustered Objects. We developed a model for Clustered Objects based on pre-existing support in Tornado and focused on operating system implementations. In particular, we designed a generic C++ class representation for Clustered Objects and implemented it on top of the Tornado operating system. This class representation is used to build two sets of example Clustered Objects. The first set consists of four different implementations of a simple integer counter. The second set consists of three different implementations of a more complex SMP caching data structure for data lookup. We evaluated

the performance of the implementations using both simulations and hardware and show that the Clustered Objects developed are able to realize performance optimizations on a shared memory multiprocessor.

Based on our experience so far, Clustered Objects are able to exploit the advantages of object-oriented technology and yet at the same time can support fine-grain SMP optimizations needed for good performance. In our examples, Clustered Object implementations are able to achieve the same performance advantages as hand-optimized implementations. We found that it is possible to create a family of Clustered Objects consisting of different implementations, each with the same external interface, but each tuned for a different access pattern. Thus it should be possible to develop a standard foundation library of Clustered Objects that will allow a client programmer to build high-performance SMP software by simply choosing the right implementations for the access pattern expected.

The remainder of this chapter briefly describes SMP architectures and operating systems, as well as the partitioned object model. The next chapter, Motivation and Background, defines locality management in the context of an SMP. The chapter also gives an example illustrating the importance of locality to performance. Chapter 3 entitled Clustered Objects, describes the Clustered Object model, Tornado support for Clustered Objects, and the class representation developed. Chapter 4 presents the two sets of example Clustered Objects implemented. Chapter 5 presents the results from the performance tests of the example Clustered Objects. Chapter 6 presents guidelines for the development of Clustered Objects. Chapter 7 proposes a new Clustered Object Model we have not yet implemented, which addresses shortcomings in the initial model.

1.1 SMPs

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

Physical limits, cost efficiency and desire for scalability have lead to SMP architectures that are

formed by inter-connecting clusters of processors. Each cluster typically contains a set of processors and one or more memory modules. The total physical memory of the system is distributed as individual modules across the clusters, but each processor in the system is capable of accessing any of these memory modules in a transparent way, although it may suffer increased latencies when accessing memory located on remote clusters. SMPs with this type of physical memory organization are called Non-Uniform Memory Access (NUMA) SMPs. Examples of such NUMA SMP architectures include Stanford's Dash [21] and Flash [17] architectures, University of Toronto's Hector [42] and NUMAchine [41] architectures, Sequent's NUMA-Q [32] architecture and SGI's Cray Origin2000 [19]. NUMA SMPs that implement cache coherency in hardware are called CC-NUMA SMPs. In contrast, multiprocessors based on a single bus have Uniform Memory Access times and called UMA SMPs.

It can be difficult to realize the performance potential of a CC-NUMA SMP. The programmer must not only develop algorithms that are parallel in nature, but must also be aware of the subtle effects of sharing both in terms of correctness and in terms of performance. The coherence protocols and distribution of physical memory add communication latencies, and explicit synchronization, needed to ensure correctness of shared data, imposes additional computation and communication overheads. Without careful layout in memory, false sharing can occur at cache line granularity. False sharing happens when independently accessed data is co-located in the same cache line. False sharing reduces the effectiveness of the hardware caches and results in the same high cache coherence overhead as true sharing.

Memory latencies and cache consistency overheads can often be reduced substantially by designing software that maximizes the locality of data accesses. Replication and partitioning of data are primary techniques used to improve locality. Both techniques allow processes to access localized instances of data in the common case. They decrease the need for remote memory accesses and lead to local synchronization points that are less contended. In experimental results we present later, we show that these techniques can lead to an improvement in performance of two orders of magnitude in some cases.

Other more course-grain approaches for improving locality in general SMP software include automated support for memory page placement, replication and migration [18, 23, 40] and cache affinity aware process scheduling [39, 24, 13, 33, 9].

1.2 SMP Operating Systems

Poor performance of the operating system can have considerable impact on application performance. For example, for parallel workloads studied by Torrellas et al., the operating system accounted for as much as 32-47% of the non-idle execution time[36]. Similarly Xia and Torrellas showed that for a different set of workloads, 42-54% of time was spent in the operating system [43], while Chapin et al. found that 24% of total execution time was spent in the operating system[6] for their workload.

To avoid the operating system from limiting application performance, it must be highly concurrent. The traditional approach to developing SMP operating systems has been to start with a uniprocessor operating system and to then successively tune it for concurrency. This is achieved by adding locks to protect critical resources. Performance measurements are then used to identify points of contention. As bottlenecks are identified, additional locks are introduced to increase concurrency, leading to finer-grained locking. Several commercial SMP operating systems have been developed as successive refinements of a uniprocessor code base. Denham et al. provides an excellent account of one such development effort [8]. However, this approach is ad hoc in nature, leads to complex systems, and provides little flexibility in that adding more processors to the system or changing access patterns may require significant re-tuning.

The continual addition of locks can also lead to excessive locking overheads. In such cases, it is often necessary to design new algorithms and data structures that do not depend so heavily on synchronization. Examples include: Software Set Associative Cache architecture developed by Peacock et al.[28] [29], kernel memory allocation facilities developed by McKenny et al.[25], fair fast scalable reader-writer locks developed by Krieger et al.[16], performance measurement kernel device driver developed by Anderson et al.[1] and the intra-node data structures used by Stets et al.[35].

The traditional approach of adding locks and selectively redesigning also does not explicitly lead to increased locality. Chapin et al. studied the memory system performance of a commercial Unix system, parallelized to run efficiently on the 64 processor-large Stanford DASH multiprocessor[6]. They found that the time spent servicing operating system data misses was three times higher than time spent executing operating system code. Of the time spent servicing operating system data misses, 92% was due to remote misses. Kaeli et al. showed that careful tuning of their operating system to improve locality allowed them to obtain linear speedups on their prototype CC-NUMA system, running OLTP benchmarks[15].

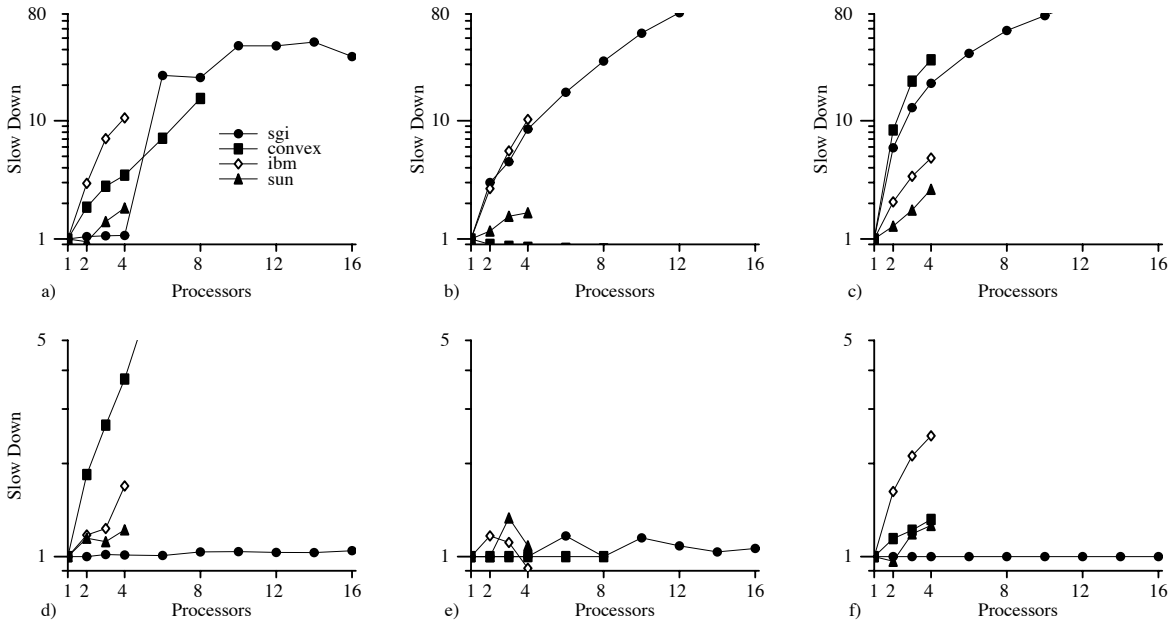


Figure 1.1: Microbenchmarks across all tests and systems. The top row (a–c) depicts the multi-threaded tests with n threads in one process. The bottom row (d–f) depicts the multiprogrammed tests with n processes, each with one thread. The leftmost set (a,d) depicts the slowdown for in-core page fault handling, the middle set (b,e) depicts the slowdown for file stat, and the rightmost set depicts the slowdown for thread creation/destruction. The systems on which the tests were run are: SGI Origin 2000 running IRIX 6.4, Convex SPP-1600 running SPP-UX 4.2, IBM 7012-G30 PowerPC 604 running AIX 4.2.0.0, Sun 450 UltraSparc II running Solaris 2.5.1.

Finally, figure 1.1 are results gathered by Gamsa et al.[12] of simple micro-benchmarks run on a number of commercial SMP operating systems. The micro-benchmarks are of three separate tests: in-core page faults, file stat and thread creation, each with n worker threads performing the operation being tested:

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

File Stat Each worker thread repeatedly fstated an independent file.

Thread Creation Each worker successively created and then joined with a child thread (the child does nothing but exit).

Each test was run in two different ways; multi-threaded and multi-programmed. In the multi-threaded case the test was run as described above. In the multi-programmed tests, n instances of the test were started with one worker thread per instance. Although the commercial systems do reasonably well on the multiprogrammed tests in general, they suffer considerable slow downs on

the multithreaded tests. This evidence implies that the existing techniques used by commercial systems are insufficient in their ability to exploit the concurrency of these simple multithreaded micro-benchmark applications.

As will be shown in chapter 3, Clustered Objects provide a framework for designing and implementing SMP software which is both highly concurrent and supports replication and partitioning of data so as to maximize locality.

1.2.1 Hive and Hurricane

The Stanford Hive operating system[7] and the University of Toronto's Hurricane operating system[37] were designed to address the locality issues in large-scale SMP operating systems. Hive focused on locality, firstly as a means of providing fault containment and secondly as a means for improving scalability. Hurricane focused on the scalability and performance aspects of increased locality. Both systems are structured as a set of individual, small-scale SMP operating system instances, which cooperatively manage the resources of the entire system.

In both approaches, the resources of the physical system are partitioned into fixed clusters containing a set number of processors and associated main memory. The resources of each cluster are managed by a separate instance of a small-scale SMP operating system. Explicit use of shared memory is only allowed within a cluster. Any co-ordination/sharing between clusters occurs using a more expensive message passing facility. It was hoped that any given request by an application could in the common case be serviced on the cluster on which the request was made with little or no interaction with other clusters.

The fixed clustering approach limits the number of concurrent processes that can contend on any given lock to the number of processors in a cluster. Similarly, it limits the number of per-processor caches that need to be kept coherent. Finally, it also ensures that each data structure is replicated into the local memory of each cluster.

One of the key observations made by the Hurricane group was that fixed cluster sizes were too restrictive. Although an attempt was made to determine the optimal configuration, it was realized that each service and its data structures required different degrees of clustering. Some data structures (i.e. Page Descriptor Index) are best shared across the entire system, while other data structures (i.e. Ready Queues) have better performance if they were replicated on a per-processor basis. This implied that greater flexibility with respect to the cluster sizes was required than was offered by the fixed clustering of Hurricane and Hive. It was concluded from the experiences with

Hurricane that the locality attributes of data structures need to be expressed and managed on a per-data structure basis, something that is addressed with Clustered Objects.

1.2.2 Commercial OSes

Many commercial vendors offer either separate SMP operating systems products or SMP versions of their uniprocessor operating systems. These include: Cellular Irix and Irix from SGI, AIX from IBM, Solaris from SUN, DYNIX/ptx from Sequent, Digital Unix from COMPAQ Digital products and HP-UX from HP. All these systems need to face the challenges of increased memory latencies with respect to CPU cycle times and as such need to optimize cache performance. Many of the vendors list scalability as an important goal in satisfying customer's need for incremental growth and protection of investments.

1.2.3 The Tornado Approach

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

More specifically, the design of Tornado is based on the observation that: (*i*) operating systems are driven by the request of applications on virtual resources, (*ii*) to achieve good performance on multiprocessors, requests to different resources should be handled independently, that is, without accessing any common data structures and without acquiring any common locks, and (*iii*) the requests should, in the common case, be serviced on the same processor they are issued on. This is achieved in Tornado by adopting an object-oriented approach where each virtual and physical resource in the system is represented by an independent object so that accesses on different processors to different objects do not interfere with each other. Details of the Tornado operating system can be found in [12, 11].

The natural outgrowth of the Hurricane experience was to build an operating system in which each data structure could specify its own clustering size. Tornado serves as the operating system for the NUMAchine multiprocessor [41]. In Tornado, unlike Hurricane, there is only one operating

system instance, but clustering is provided for on a per-object basis. Clustered Objects are used for this purpose. Tornado is implemented in C++ using an object oriented structure.

At this point, the majority of the system's objects are naive Clustered Objects using just a single representative. The majority of work performed to date has been on developing the underlying infrastructure and basic functionality needed for Clustered Objects. Tornado is now at the point where the Clustered Object model can be more formally developed and the current system Clustered Object implementations replaced with more advanced implementations tuned for performance.

1.3 Partitioned Objects

In a partitioned object model, a single partitioned object with a well-defined external interface can be decomposed into multiple, more elementary objects, called representatives. Each representative acts on behalf of the entire partitioned object, servicing the requests from a restricted local access domain. A local access domain is a subset of the physical resources of the system from which the partitioned object can be accessed. For example, in a distributed environment, local access domains would be individual machines on the network and in a more tightly coupled environment, local access domains would be individual processors. The representatives of a partitioned object can interact and cooperate internally, if necessary, using the communication facilities of the environment. Distributed Shared Objects [38, 14] and Fragmented Objects [22, 5] are the only partitioned object models other than Clustered Objects, that we are aware of. Both are designed for distributed environments.

Distributed Shared Objects are designed as a framework for developing wide-area distributed applications. The state of a Distributed Shared Object can be, at the same time, physically distributed across multiple machines. The distributed nature of a Distributed Shared Object is hidden from clients behind its interface. Communication between the representatives of a Distributed Shared Object are implemented on top of standard wide area network protocols. The most important design issue addressed by Distributed Shared Objects is scalability in the context of the World Wide Web [38].

The Fragmented Objects framework also targets the development of distributed applications. However, Fragmented Objects are aimed at local area networks. Similar to Distributed Shared Objects, a Fragmented Object appears to its clients as a single entity defined by its external

interface. Internally, a Fragmented Object encapsulates a set of cooperating representatives that use standard network protocols for communication between representatives.

Both models are aimed at coarse-grained performance optimizations and at managing the complexity of networked environments. In contrast, Clustered Objects target SMP's, where the performance tradeoffs are considerably different than in a distributed environment.

1.3.1 Clustered Objects

A Clustered Object is identified by an address space unique identifier. The identifier locates a per-processor representative object for the Clustered Object. All accesses to a Clustered Object on a processor are directed to a specific representative. To allow for more efficient use of resources, the representatives of a Clustered Object can be instantiated on first use. All the representatives of a Clustered Object are managed via a special per-Clustered Object management object. The management object is responsible for instantiation, deletion and assignment of representatives to processors. A Clustered Object can have a single shared representative that is assigned to all processors, a representative per-processor or any a configuration in between. Chapter 3 gives a detailed description of Clustered Objects.

1.4 Torando support for Clustered Objects

An operating system infrastructure is needed to implement Clustered Objects efficiently. In Tornado this includes:

- Object Translation Facility
- Kernel Memory Allocation Facility (KMA)
- Protected Procedure Call Facility (PPC)

1.4.1 Object Translation Facility

The Object Translation Facility of Tornado is used to locate the processor-specific representative object when a Clustered Object is accessed on a given processor. It is implemented with two sets of tables per address space, a global table of pointers to per-Clustered Object management objects, and per-processor tables of pointers to representatives. The identifier for a Clustered Object is a common offset into the tables. If no representative exist for a given processor the global table is

consulted to locate the Clustered Object's management object that manages all the representatives of the Clustered Object. Details of the Object Translation Facility is provided in Chapter 3.

1.4.2 KMA

The Kernel Memory Allocation facility manages the free pool of global and per-processor memory. It is capable of allocating memory from pages that are local to a target processor. By overloading the default new operator with a version that calls the localized memory allocation routines of the Kernel Memory Allocation facility, Tornado ensures that default object instantiation occurs with processor local memory. Hence, representatives and the data they allocate, automatically reside on the processors on which they are instantiated. This helps to reduce false sharing across clusters.

1.4.3 PPC

The Protected Procedure Call facility of Tornado supports interprocess communication. Protected Procedure calls allow one process within an address space to invoke the methods of an Object in another address space. A Protected Procedure Call is implemented as a light-weight protection domain crossing, executed on the same processor from which they are called. The Protected Procedure Call facility also provides the ability for a process executing on one processor, to invoke a procedure to be executed on another processor within the same address space, although at higher cost. This form of cross-processor Protected Procedure Calls will be referred to as *Remote Procedure Calls*. Clustered Objects can use Remote Procedure Calls to implement function shipping as another form of cooperation between representatives.

Chapter 2

Background and Motivation

2.1 Background

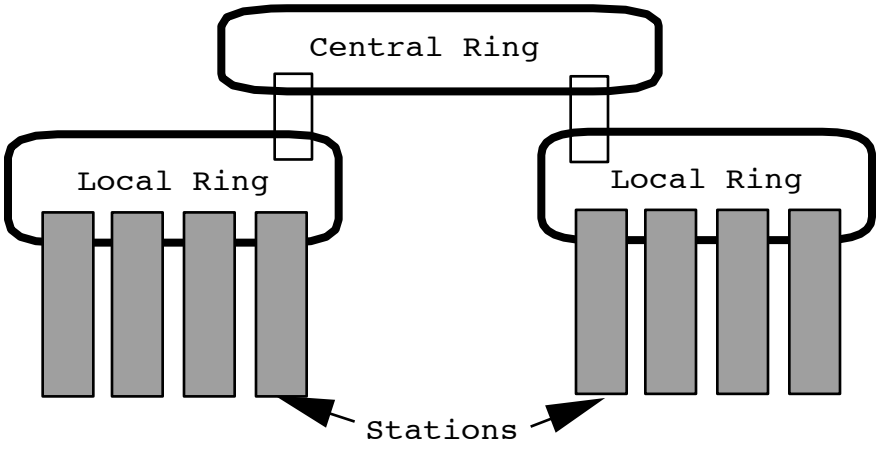
In this dissertation, we are specifically interested in the attributes of NUMA architectures. As a case in point, we present an overview of the NUMAchine hardware architecture in section 2.1.1. Section 2.1.2 then discusses the software issues of NUMA SMP architectures.

2.1.1 NUMAchine Architecture

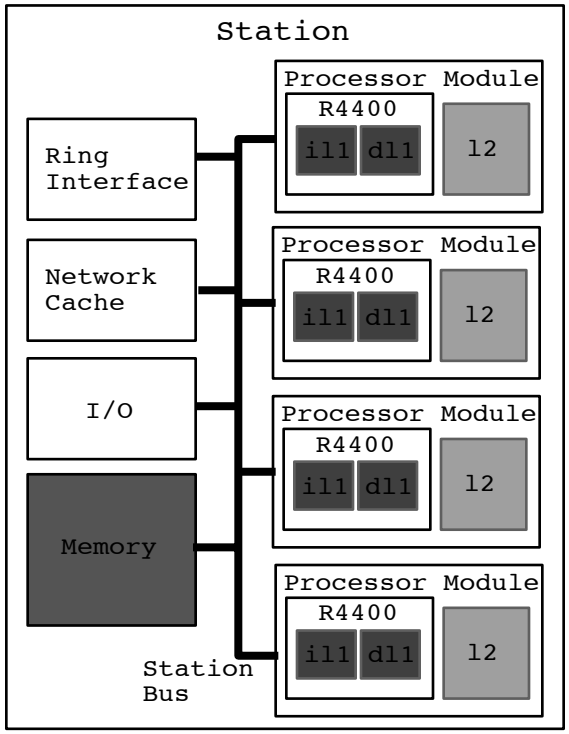
NUMAchine is a NUMA shared memory multiprocessor, consisting of interconnected stations, each composed of several processors, memory modules, and I/O capabilities[1]. The physical memory is thus distributed across the stations. A flat physical addressing scheme is used, with a specific address range assigned to each station. All processors access all memory locations in the same manner. The time needed by a processor to access a given memory location depends upon the distance between the processor and the memory.

NUMAchine uses a ring-based hierarchical interconnection network. At the lowest level of the hierarchy, stations contain several processors connected by a bus as shown in the bottom portion of figure 2.1. A processor module contains a processor with on-chip, level 1 data and instruction caches, and an external level 2 secondary cache. The stations are interconnected by bit-parallel rings, as shown in the top portion of figure 2.1. For simplicity, the figure shows only two levels of rings — *local* rings connected by a *central* ring. Our planned hardware prototype machine will have 4 processors in each station; 4 stations per local ring and 4 local rings connected by a central ring.

[1]See Vranesic et al. for a detailed presentation of the NUMAchine architecture [41].



NUMAchine Ring Hierarchy



Station Organization

Figure 2.1: NUMAchine Architecture.

The largest configuration we consider in this study is a 16-processor system, composed of one local ring, with 4 stations each containing 4 processors.

The NUMAchine memory hierarchy consists of four levels. Each processor has a level 1 cache on-chip and an external secondary cache. The next level consists of the memory located in the same station. This includes the memory module(s) for the physical address range assigned to the station, and the station's network cache, which is used as a cache for data whose home memory is in a remote station. The final level in the memory hierarchy consists of all memory modules that are in remote stations.

NUMAchine includes a hardware-supported cache coherence protocol that is efficient and inexpensive to implement [41]. The coherence protocol takes advantage of the ordering property of the NUMAchine hierarchy to optimize performance. In particular, a single message suffices to perform multiple invalidations and no acknowledgments are required.

To help generalize the results of this study our performance study also includes results from simulations that do not take advantage of any of NUMAchine's unique features such as the network caches or its optimized cache coherence protocol.

2.1.2 Software Issues

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

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

Cache Misses: Efficient use of caches is critical to good performance for two reasons. Firstly, a low cache miss rate ensures that processors do not spend large amounts of time stalling on memory accesses. Secondly, it reduces the traffic on shared system busses. With per-processor caches, processors accessing data on the same cache line, either because the data

is being shared directly or because it is being shared falsely[2], causes the line to be replicated into multiple caches. Sharing of cache lines causes an increase in consistency overhead and cache misses[3]. Avoiding shared data and carefully laying out data in memory to avoid false sharing can reduce cache line sharing and the associated increase in overheads substantially.

Remote Memory Access: to achieve good performance, the extra costs of remote memory accesses must be avoided. Caches can help to reduce the cost of remote accesses, but do not eliminate the costs completely. The first access to a remote data element must pay the extra costs. Additionally, true and false read/write sharing can force invalidation of locally cached copies of remote data. Avoiding of shared data and carefully placing data in the memory modules closest to the processors that access the data can reduce the number of remote memory accesses.

The term *locality management* refers to the combination of increasing concurrency, reducing cache misses and reducing remote memory accesses. Gamsa et al. have outlined a set of design principles for developing software that manages locality [10], the main points of which are:

- Concurrency
  - Replicate read locks and implement write locks as a union of the read locks. This increases concurrency by making the locks finer grained.
- Cache Misses
  - Segregate read-mostly data from frequently modified data to reduce misses due to false sharing.
  - Segregate independently accessed data to eliminate false sharing.
  - Replicate write-mostly data to reduce sharing.
  - Use per-processor data wherever possible to avoid sharing.
  - Segregate contended locks from their associated, frequently modified data. This avoids lock contenders interfering with the lock holder.
  - Co-locate un-contended locks with their associated data to better utilize spatial locality and reduce the number of cache misses.
- Remote Memory Accesses

[2]False sharing is the accessing of different data elements that happen to reside on the same cache line.

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

  - Ensure read-mostly data is replicated into per-processor memory.
  - Migrate read/write data between per-processor memory if accessed primarily by one processor.
  - Replicate write-mostly data where possible and ensure replicas are in per-processor memory.
- Algorithmic
  - Use Approximate local information rather than exact global information.
  - Avoid Barriers

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

2.2 Motivating Example

This section will present a simple example to illustrate why locality management at the individual data object level is important. The implementation of a simple integer counter on the NUMAchine hardware will serve as a running example. The following subsections describe the scenario in which the counter will be used, an abstract data type for the counter, and four different implementations. The performance of the four implementations will be presented to motivate the advantages of locality management.

2.2.1 Scenario

We will assume that there is a pair of high frequency events, A and B, which we desire to count. When A occurs we would like to increment the counter and when B occurs we would like to decrement the counter. Both events occur independently on all processors of the system. The value of the counter will be read infrequently compared to the frequency of the events: only 1% of all accesses to the counter will be to read its value. Moreover, it is unnecessary for the counter to return an exact value on reads, as an approximate value suffices. Such a counter might be used to gather performance statistics on the average queue length for a system resource.

```
Type:
   Counter
operations:
   value()     : return current integer value of counter
   increment() : adds 1 to the current value of the counter
   decrement() : subtracts 1 from the current value of the counter
constraints:
   Given: Counter c
   Initially c.value()=0
   (c.increment()).value = c.value() + 1
   (c.decrement()).value = c.value() - 1
```

Figure 2.2: Definition of the Counter Abstract Data Type

```
class integerCounter {
public:
    virtual void value(int &val)=0;
    virtual void increment()    =0;
    virtual void decrement()    =0;
};
```

Figure 2.3: C++ interface definition for the Counter ADT.

2.2.2 Abstract Data Type

Figure 2.2 presents a trivial abstract data type Counter. *Increment* and *decrement* operations modify or write the counter the *value* operation reads the counter.

Following good C++ practice, an abstract base class, *integerCounter*, is used to specify the interface for all integer counters and is presented in figure 2.3. This ensures that all the integer counter implementations will be interchangeable.

2.2.3 Implementations

We consider four implementations of *integerCounter*: *SharedCounter*, *CounterArray*, *CounterArrayPadded*, and *CounterLocalized*, each refining the implementation of the previous.

Shared Counter

The simplest implementation for the counter is to use a single shared integer variable as illustrated in figure 2.4. The *increment* and *decrement* methods use atomic update primitives to ensure proper synchronization.

The left-most bar of each set of bars of figure 2.5 illustrates the performance of the *SharedCounter*. The performance illustrated is from a simple test in which a total of 4096 requests are made to the counter with 1% being invocations of the value method, and the remaining 99% being

```
class SharedCounter : public integerCounter {
    int _count;
public:
    SharedCounter()              { _count=0; }
    virtual void value(int &val) { val=_count; }
    virtual void increment()     { FetchAndAdd(&_count,1); }
    virtual void decrement()     { FetchAndAdd(&_count,-1); }
};
```

Figure 2.4: C++ Shared Counter implementation.

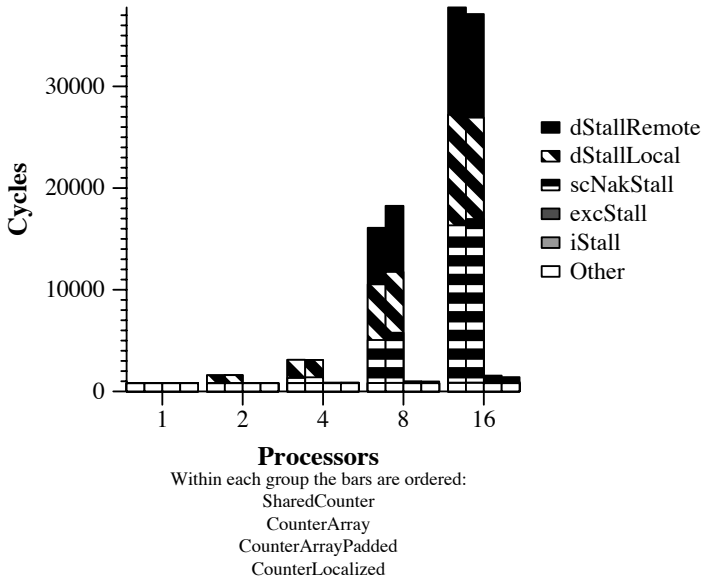


Figure 2.5: Performance results for integer Counter implementations.

evenly divided between invocations of *increment* and *decrement*. The graph shows the performance on 1, 2, 4, 8, and 16 processors with the y-axis plotting the average number of cycles required per request. The total number of requests are divided evenly across the number of processors in the test. Chapter 5 provides more details into the experimental setup.

The cycles per request increases more than linearly as more processors are used. The increases are primarily due to the increase in coherence overhead and cache misses as more processors participate. With one processor participating, the counter remains in the cache, but with multiple processors participating, each modification at one processor, causes the cached value to be invalidated at each other processor, causing subsequent cache misses. This performance behaviour is clearly undesirable. It costs two orders of magnitude more per request on 8 and 16 processors than on one processor.

```
class CounterArray : public integerCounter {
    int *_count;
public:
    CounterArray()               {
                                    _count=new int[NUMPROC];
                                    for (int i=0;i<NUMPROC;i++)
                                      _count[i]=0;
                                 }
    ~CounterArray()              { delete[] _count; }
    virtual void value(int &val) {
                                   val=0;
                                   for (int i=0;i<NUMPROC;i++)
                                     val+=_count[i];
                                 }
    ~CounterArray()              { delete[] _count; }
    virtual void increment()     { FetchAndAdd(&(_count[MYVP]),1); }
    virtual void decrement()     { FetchAndAdd(&(_count[MYVP]),-1); }
};
```

Figure 2.6: C++ Code for CounterArray class. The *MYVP* macro expands to a unique index that can be used to identify the processor on which the code is executing.

Counter Array

To try and improve on the performance of the *SharedCounter* implementation, the *CounterArray* implementation (illustrated in figure 2.6) partitions the counter into an array of counters. Each processor is given its own counter within the array. This should allow each processor to modify its own local counter without interfering with other processors.

When the *CounterArray* is instantiated, its constructor allocates an array of counters, one per processor in the system. The *increment* and *decrement* methods now only modify the counter of the processor on which the operations are invoked. The *value* sums the value of all the local counters to yield the total value.

The *value* method of *CounterArray* is not atomic with respect to the updates. As a result, while the *value* method sums each per-processor counter, the values may be changing, and thus the value returned does not necessarily correspond to the value of the counter at time of invocation. A more synchronous approach would be possible by adding a lock to each per-processor counter and acquiring and releasing it for modifications. The *value* method could then globally lock the counter by first acquiring all the per-processor locks. However, the scenario presented in 2.2.1 only requires an approximate value of the counter, so the *CounterArray* implementation with no locks is more efficient.

The second bar in Figure 2.5 illustrates the performance of *CounterArray*. Note that the

```
class CounterArrayPadded : public integerCounter {
    struct counter {
        int val;
        char pad[SCACHELINESIZE - sizeof(int)];
    } *_count;
public:
    CounterArrayPadded()         {
                                    _count=new struct counter[NUMPROC];
                                    for (int i=0;i<NUMPROC;i++)
                                        _count[i].val=0;
                                 }
    ~CounterArrayPadded()              { delete[] _count; }
    virtual void value(int &val) {
                                   val=0;
                                   for (int i=0;i<NUMPROC;i++)
                                     val+=_count[i].val;
                                 }
    virtual void increment()     { FetchAndAdd(&(_count[MYVP].val),1); }
    virtual void decrement()     { FetchAndAdd(&(_count[MYVP].val),-1); }
};
```

Figure 2.7: C++ code that implements CounterArrayPadded class.

performance is no better than that of the *SharedCounter*. Although *CounterArray* partitions the counter and avoids the data sharing present in the *SharedCounter* implementation, it does not avoid false sharing. Since each integer value is 4 bytes and a secondary cache line of the machine is 128 bytes, 16 counters easily fit on one cache line. This means that each time a counter is updated, the updating processor will still interfere with all other processors, as was the case with *SharedCounter*.

Counter Array Padded

The addition of padding to the per-processor counters can eliminate the false sharing in *CounterArray*. Figure 2.7 illustrates such an implementation.

The performance of *CounterArrayPadded* has a marked improvement over *SharedCounter* and *CounterArray*, as illustrated in figure 2.5. The elimination of true and false sharing means that modifications to the counters on each processor can occur concurrently and without cache interference. This results in an improvement of two orders of magnitude on 8 and 16 processors.

Counter Localized

Although *CounterArrayPadded* eliminates all sharing, it does not ensure that the per-processor counters are located in the memory modules closest to the processors accessing them. Figure 2.8 presents an implementation that ensures that each per-processor counter is located in the memory

```
class CounterLocalized : public integerCounter {
    struct counter {
        int val;
        char pad[SCACHELINESIZE - sizeof(int)];
    } **_count;
public:
    CounterLocalized()           {
                                    StubXAppl appl(myAppl->getOH());
                                    _count=new struct counter *[NUMPROC];
                                    for (int i=0;i<NUMPROC;i++) {
                                        if ( i != MYVP )
                                        {
                                            appl.setVP(i);
                                            appl.createProcess1AndWait(
                                                (tstatusfunc)doremoteinit,
                                                (reg_t)this);
                                        } else
                                            init();
                                    }
                                 }

    virtual void init()          {
                                    _count[MYVP]=new struct counter;
                                    _count[MYVP]->val=0;
                                 }

    ~CounterLocalized()          {
                                    for (int i=0;i<NUMPROC;i++)
                                        delete _count[i];
                                 }
    virtual void value(int &val) {
                                   val=0;
                                   for (int i=0;i<NUMPROC;i++)
                                          val+=_count[i]->val;
                                 }
    virtual void increment()     { FetchAndAdd(&(_count[MYVP]->val),1); }
    virtual void decrement()     { FetchAndAdd(&(_count[MYVP]->val),-1); }
};

void doremoteinit( reg_t obj)
{
    ((CounterLocalized *)obj)->init();
}
```

Figure 2.8: C++ code that implements CounterLocalized class.

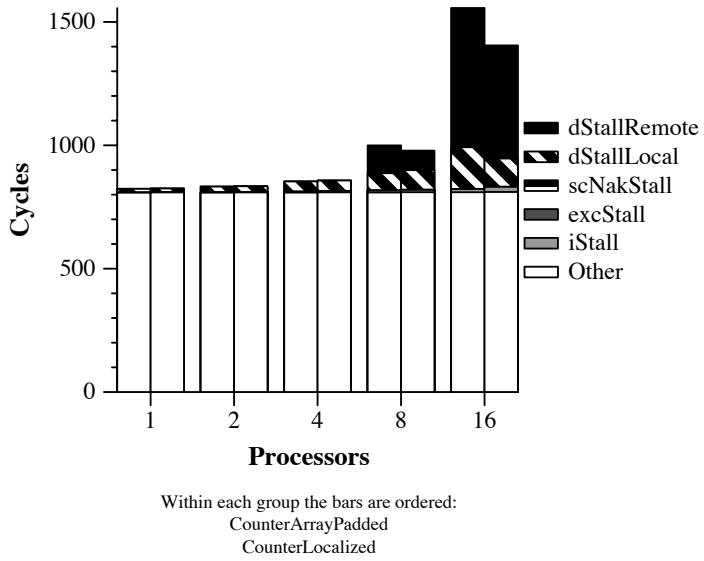


Figure 2.9: Performance Results for the CounterArrayPadded and CounterLocalized Implementations.

module closest to the processor accessing it.

Tornado's memory allocator ensures that memory allocations made on a processor are satisfied by memory pages from the station the processor belongs to. The *CounterLocalized* exploits this feature to locate each per-processor counter correctly. Rather than maintaining an array of counters, the *CounterLocalized* implementation keeps an array of pointers to counters. When *CounterLocalized* is instantiated, its constructor uses a remote procedure facility to invoke the *init* method on each processor of the system. The *init* method allocates a per-processor counter and records a reference to it in the array of pointers. The *increment* and *decrement* methods dereference the appropriate pointer within the array to yield the right per-processor counter. The *value* method similarly dereferences each pointer within the array to yield the actual counters.

Figure 2.5 illustrates that the performance of *CounterLocalized* is similar to that of *CounterArrayPadded*. Figure 2.9 compares the performance of the *CounterArrayPadded* and the *CounterLocalized* implementations alone. It shows that *CounterLocalized* performs slightly better; there is a drop in time spent stalling on remote data. It is not possible to eliminate all remote memory accesses, as 1% of all requests are invocations of the *value* method, which requires remote memory accesses by definition. While the difference in performance between *CounterArrayPadded* and *CounterLocalized* is not large, one should keep in mind that the multiprocessor on which the experiments were performed is relatively small; the difference will be larger on larger systems or on

systems where the cost of remote memory accesses is larger relative to the cost of local accesses[4].

2.2.4 Summary

The performance of the different integer counter implementations show how careful locality management can yield a significant performance improvement. As will be shown in the next chapter, Clustered Objects are designed to make it easier to implement locality management on a per-object basis.

[4]A remote uncached and uncontended memory access is approximately 3.5 times that of a local memory access

Chapter 3

Clustered Objects

3.1 What they are

Clustered Objects extend traditional objects so that it is possible to provide multiprocessor optimizations while maintaining a common object-oriented interface and were first described by Parsons et al. in [27]. While Object Oriented technology provides for clear separation between interface and implementation through encapsulation and information hiding — it is easy to replace one implementation with another, without affecting the clients of a given object — traditional Object Oriented approaches do not provide any standard means for implementing multiprocessor optimizations behind a fixed interface. Clustered Objects provide exactly that.

A Clustered Object appears externally as a regular (C++ like) object to its clients. Internally, however, it is constructed out of one or more representative objects, each associated with a specific subset of processors. Clustered Objects share three important aspects in common with standard objects:

1. a single, well-defined interface;
2. a unique reference for identifying each instance; and
3. an internal structure that is completely hidden from clients.

The unique features of Clustered objects are:

1. Internally, the representative objects that implement a Clustered Object cooperate to replicate, partition and/or migrate the data with the goal of increasing locality.

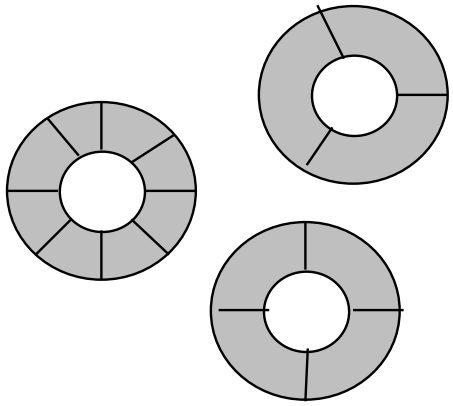


Figure 3.1: Abstract view of traditional object-oriented system.

2. Client accesses to the Clustered Object are transparently directed to a local point of access, namely the internal representative object associated with the processor on which the access is being made.

The internal *representatives* objects are standard C++ objects and are typically instantiated on first use[1]. Together, the representatives of a Clustered Object implement the functionality of the Clustered Object. Representatives are free to share and cooperate by any means available, including the use of shared memory and remote procedure calls. It is up to the implementor to maximize locality and minimize global interaction whenever possible.

The next section outlines the details of the Clustered Object model. It is followed by a section which details the internal system mechanisms of Tornado that support Clustered Objects. The last section describes the class representation that was implemented for the development of Clustered Objects.

3.2 The Clustered Object Model

This section describes the general Clustered Object model, which was based on the Tornado operating system's support for partitioning objects within an address space.

The Clustered Object model is a partitioned object model for shared memory multiprocessors. In a traditional, object-oriented model, a software system is designed as a set of well-defined independent objects. Encapsulation, information hiding and separation between interface and implementation are key features in this model. Every object exports an "external" interface to the

[1]It is also possible to instantiate all reps when the CO is instantiated.

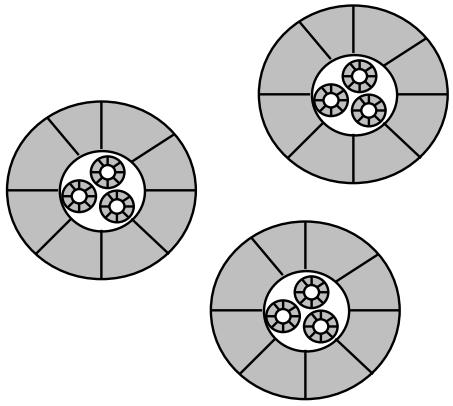


Figure 3.2: Abstract view of a Clustered Object system.

other objects in the system, completely hiding their internal structures. Figure 3.1 illustrates an abstract view of a system composed of three objects. Each object has an external interface which is represented by the shaded portion. The external interface is composed of individual methods that can be invoked by other objects in the system and are represented as the partitions in the shaded portion. At the core of each object is its internal data, represented by the unshaded part at the center of each object. The methods of an individual object can access its internal data, but methods of other objects cannot.

The Clustered Object model also adheres to this object-oriented view, but adds an extra level of structure to accommodate locality issues that arise in SMPs. Traditional object-oriented programming does not guide the internal structuring of objects in any way. In contrast, Clustered Object programming, suggests structuring the internal data as a collection of representative objects. Figure 3.2 illustrates this view. Each representative is assigned to handle the requests from a subset of processors in the system. The model advocates that representatives be implemented to handle all invocations of the Clustered Object's externally visible methods. Requests should be handled locally by the representative whenever possible, and global interaction between representatives should be used only when necessary and done transparently to the clients. This encourages implementing the internal structures of Clustered Objects in a distributed manner, stressing locality.

Figure 3.3 illustrates this internal view. Note that the representatives are illustrated as data instances which are associated with a given cluster of processors, and that all representatives are accessed via an interface composed of methods as defined by the Clustered Object.

A number of potential organizations and policies for the structuring of representatives within a Clustered Object exist. The next few paragraphs highlight some of the options available and

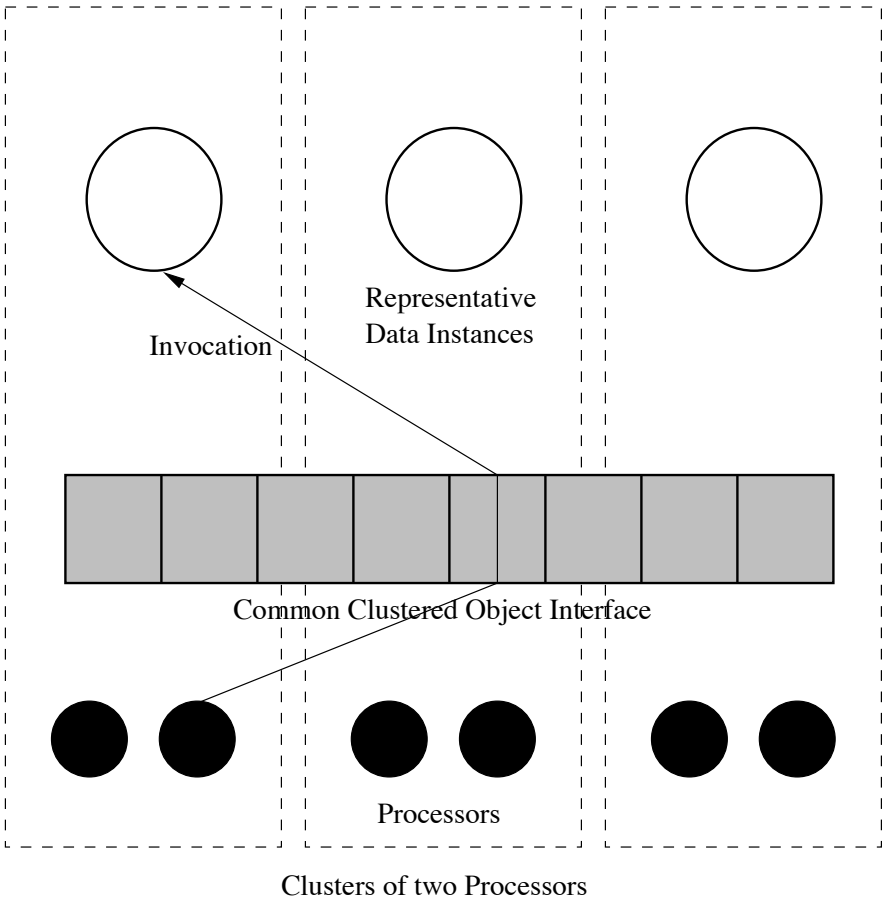


Figure 3.3: Internal abstract view of a Clustered Object. Each dash-lined box represents a cluster of processors. The filled circles represent processors. Unfilled circles are the individual representatives assigned to each cluster. All representatives share a common interface. Invocations of a method of the Clustered Object on a processor indirectly invokes the corresponding method of the representative.

identify which ones we focus on.

The model makes no restrictions in assigning representatives to processors; as such, a range of potential organizations are possible from one representative per Clustered Object to one representative per processor. The maximum number of processors assigned to any one representative is called the *clustering factor* or *degree of clustering* (see figure 3.4). While it is possible to define Clustered Objects with representatives being assigned different numbers of processor this work will focus only on clustered objects with fixed degrees of clustering.

The model does not require that representatives be instances of the same class, although they usually are. The only restriction is that all representatives export the external interface of the Clustered Object. We will only consider the case in which all representatives are of the same class.

A natural aspect of the model is the notion of management policies for the data of the Clustered Object. Four obvious policies are: Share, Replicate, Partition and Migrate, as illustrated in figure 3.5. This work will focus on sharing, replication and partitioning. Many of the locality management optimizations involve the application of these policies. For example, replication and partitioning can be used to increase concurrency, reduce cache line sharing, localize data and segregate data.

The Clustered Object model introduces new aspects for a programmer to consider. The programmer not only has to implement the functionality of the object as defined by its external interface, but must also manage the representatives themselves, including representative creation, keeping the representative data consistent, the mapping of representatives to processors and representative destruction.

Typically, it is not known how many representatives will be needed when a Clustered Object is instantiated. It would be wasteful, for example, to instantiate a representative for each processor in the system when the application will only run on four processors. For this reason, representatives are typically instantiated on demand, when they are first needed.

In our implementation, each Clustered Object contains a management object that centralizes the management of the representative. Using a separate object allows the use of inheritance to simplify the programmer's task, and a class hierarchy of standard management policies could be provided. In Tornado's Object Translation System, the management object is called the *Miss-Handling Object* for reasons that will become clear in the next section. Figures 3.6 and Figure 3.7 illustrate Clustered Objects that include a Miss-handler. In both figures, the object with the lighter-shaded external interface represents the Miss-Handling object. Figure 3.6 shows that the Miss-Handling object is internal to the Clustered Object but is separate from the representatives.

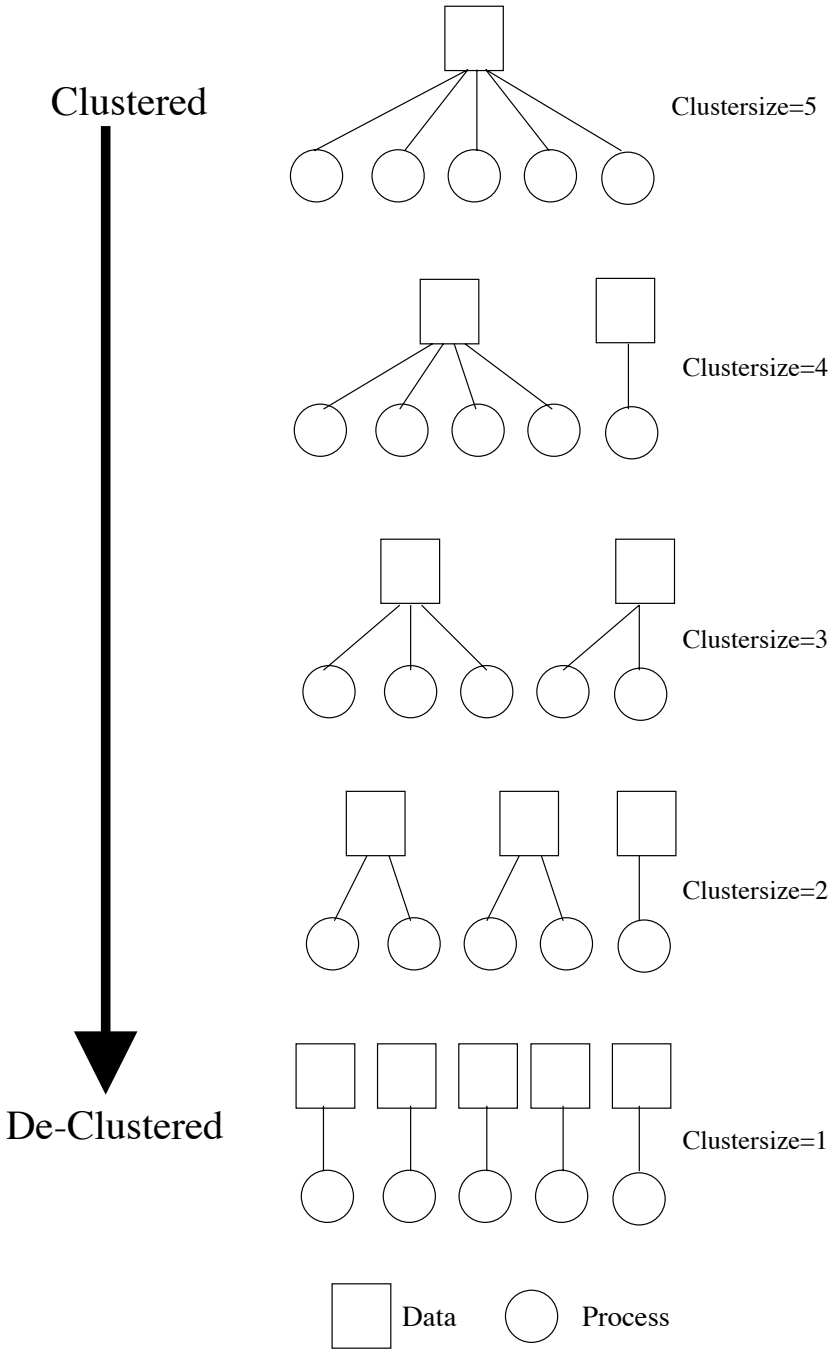


Figure 3.4: Clustering/De-Clustering Spectrum

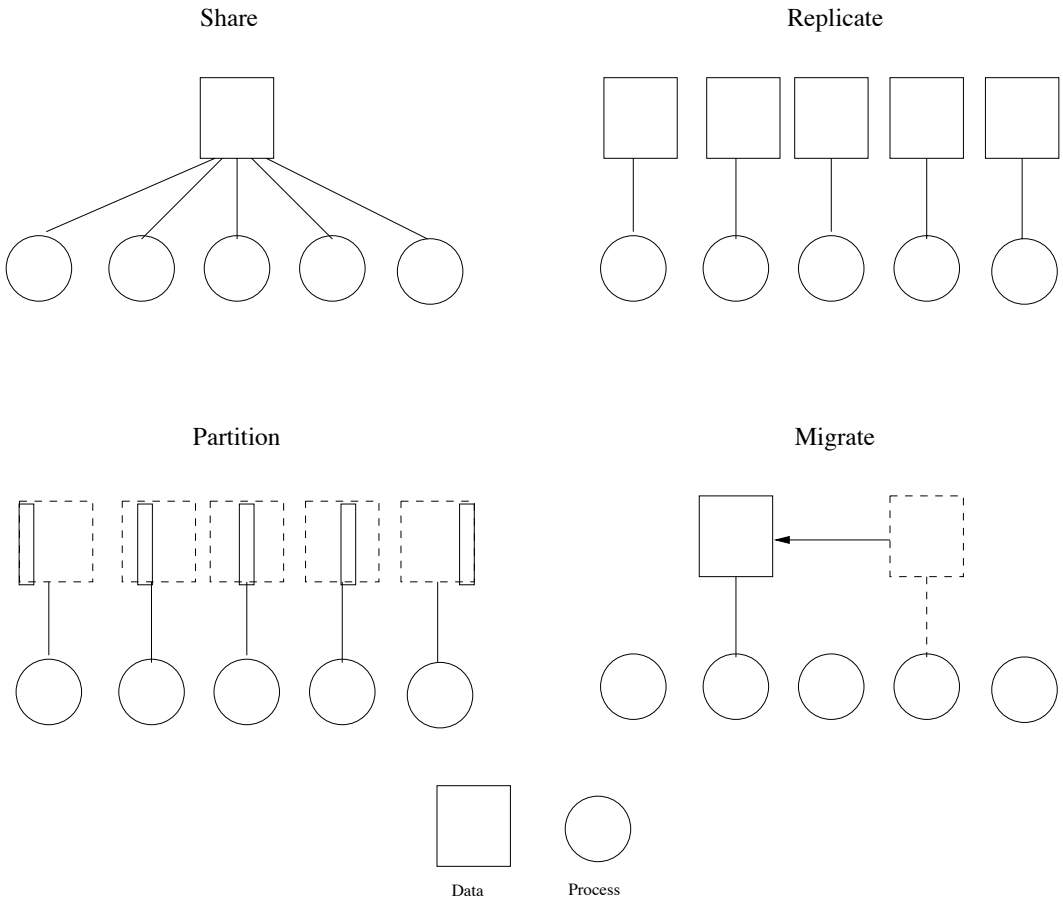


Figure 3.5: Four different Data Management Policies

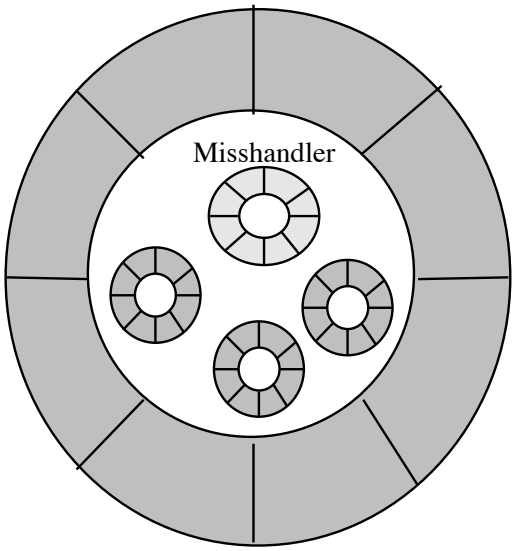


Figure 3.6: Abstract view of a Clustered Object with a MissHandler

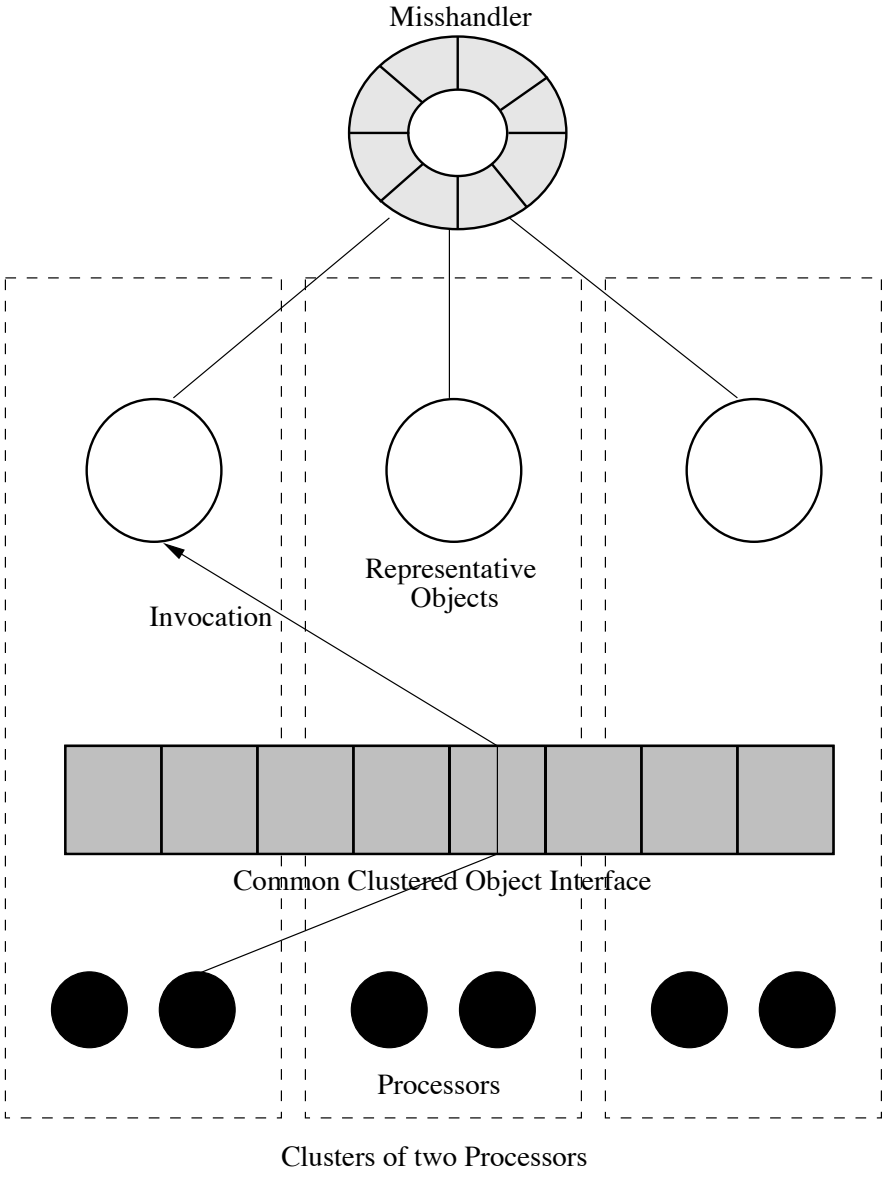


Figure 3.7: Internal Abstract view with MissHandler

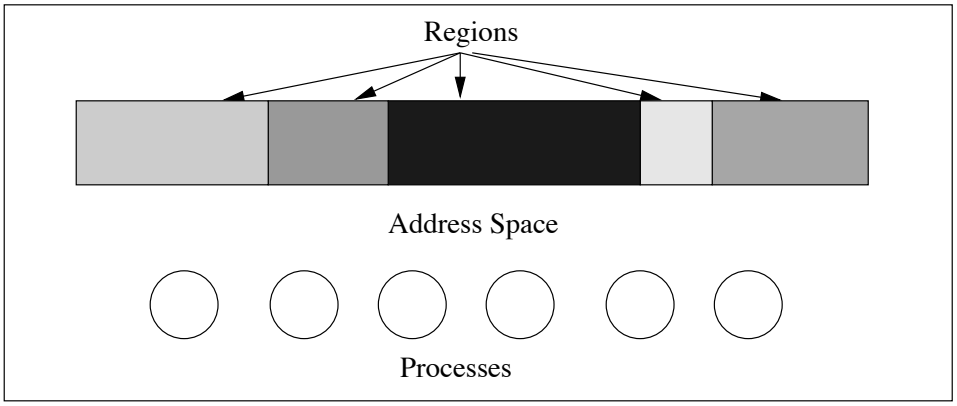


Figure 3.8: A View of a Program in Tornado

Figure 3.7 illustrates that the Miss-Handler is global to the representatives and each representative is associated with the Miss-Handler.

3.3 Tornado Support for Clustered Objects

Within Tornado, there exists a number of facilities that allow for the efficient implementation of Clustered Objects. These are:

1. a global identification mechanism for Clustered Objects.
2. a facility for associating a representative of a Clustered Object to a processor.
3. a facility for mapping a global Clustered Object identifier to the appropriate local representative, given a specific processor.
4. facilities for allocating resources local to a specific processor.
5. a facility for sharing and communicating between representatives.

In Tornado, the *Object Translation System*, provides for the first three facilities and are discussed in the following subsections. The fourth is provided for by the basic Kernel Memory Allocation facilities (KMA) of Tornado. The Protected Procedure Call facilities (PPCs) of Tornado provides explicit cross processor communications beyond the basic shared memory provided by the hardware.

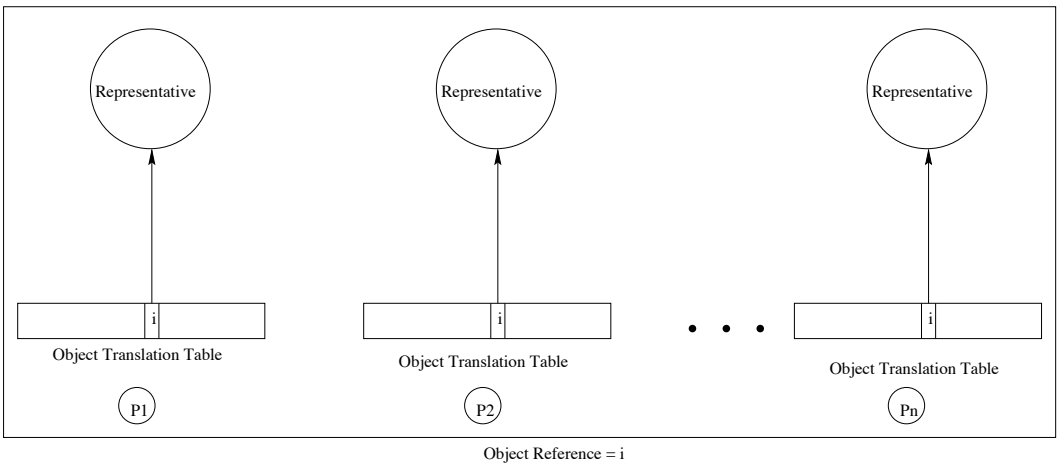


Figure 3.9: Object Translation Table organization, with one representative per processor.

3.3.1 Object Translation System

It is useful to first highlight some of Toranado's basic components. The main unit of organization is a program. A program has associated with it an address space and processes. Processes are the basic units of execution. All the processes of a program share the same address space. An address space can be broken into arbitrary regions. Each region can have its own memory management policy. Figure 3.8 illustrates these components and how they are related.

The Object Translation System provides support for implementing Clustered Objects composed of local representative objects within an address space. Each representative satisfies method invocations from processes running on a given sub-set of processors. To understand how this is achieved, we will first look at how a Clustered Object method invocation is translated to the invocation of a specific representative method on one processor.

The basic technique used, extends the standard C++ model of an object with an extra level of indirection. A Clustered Object is identified by a pointer to a pointer of a given object type, and thus accesses to the methods of a Clustered Object require two dereferences. The first dereference abstractly identifies a specific instance of a Clustered Object; in our implementation a *Clustered Object Identifier* points into a table of pointers, and each pointer in the table identifies a specific representative. The table of pointers is called an Object Translation Table. Each processor has its own Object Translation Table, so the pointers therein point to processor-specific representatives for the Clustered Objects. Method invocation is carried out after dereferencing a pointer in this table. Thus, a Clustered Object method invocation effectively invokes the corresponding method of the identified representative.

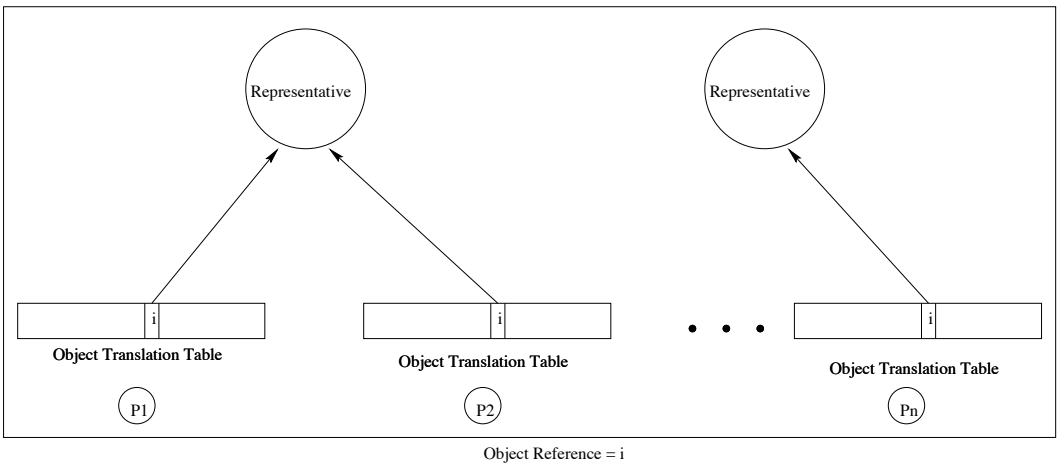


Figure 3.10: Object Translation Table organization, with one representative for every two processors.

To allow Clustered Object invocations on each processor in exactly the same way, the virtual memory capabilities of Tornado are exploited. Per-processor aliased virtual memory regions are used within the address space to give each processor its own unique copy of the Object Translation Table in an aliased memory region that is located at the same virtual address for each processor. This allows the Object Translation Table to identify the local representatives for all Clustered Objects on a given processor, and each processor can dereference the table in exactly the same way. Figure 3.9 illustrates the Object Translation Table organization with one representative per processor, and figure 3.10 shows an example in which two processors share a representative.

When a Clustered Object is created, the Object Translation System must be consulted to allocate a new Clustered Object Identifier. The Object Translation System controls the assignments of Clustered Object Identifiers to ensure that their allocation is unique across all processors. For example, if a new Clustered Object is created on one processor, the Clustered Object Identifier assigned to it must be considered allocated on all other processors to avoid conflicts. To achieve this, each processor is assigned a unique portion of the entire range of Clustered Object Identifiers. Clustered Objects created on a given processor are assigned identifiers from the processor's unique range, ensuring that the assigned identifier will not conflict with allocations on other processors. This approach avoids the need to explicitly coordinate allocations across processors.

The locality of an access is a key aspect of this design. The approach used to locate representatives avoids accesses to non-local memory in the common case, so a Clustered Object can be accessed without introducing any sharing. As a result, any locality provided by a Clustered Object is not impacted (i.e. negated) by inherent sharing in the Object Translation System.

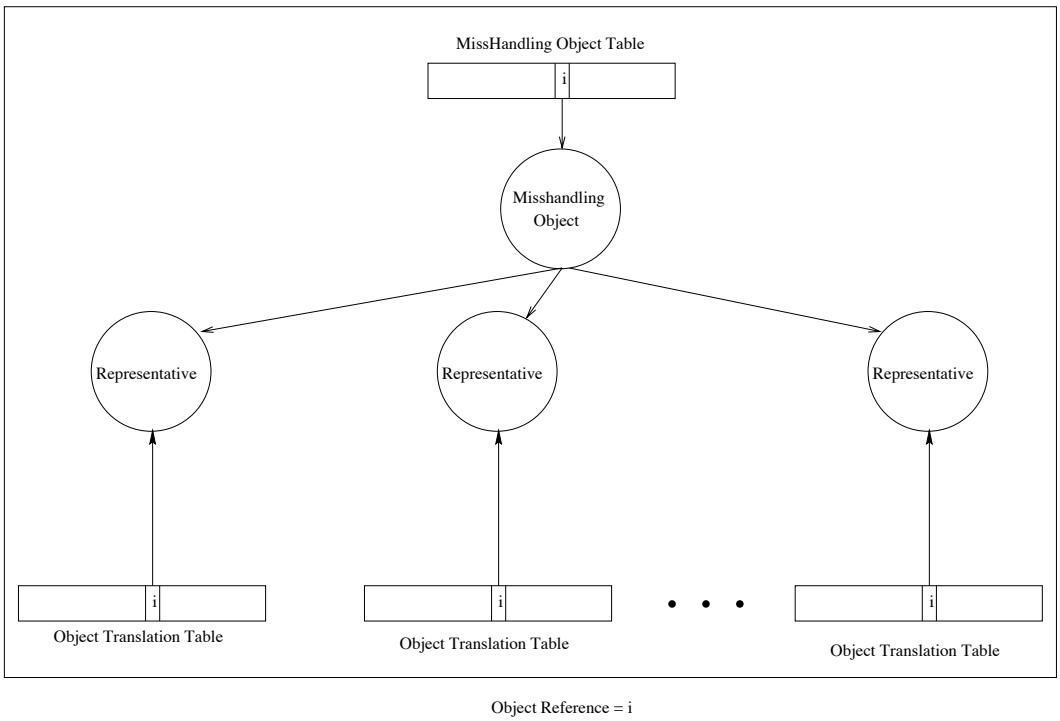
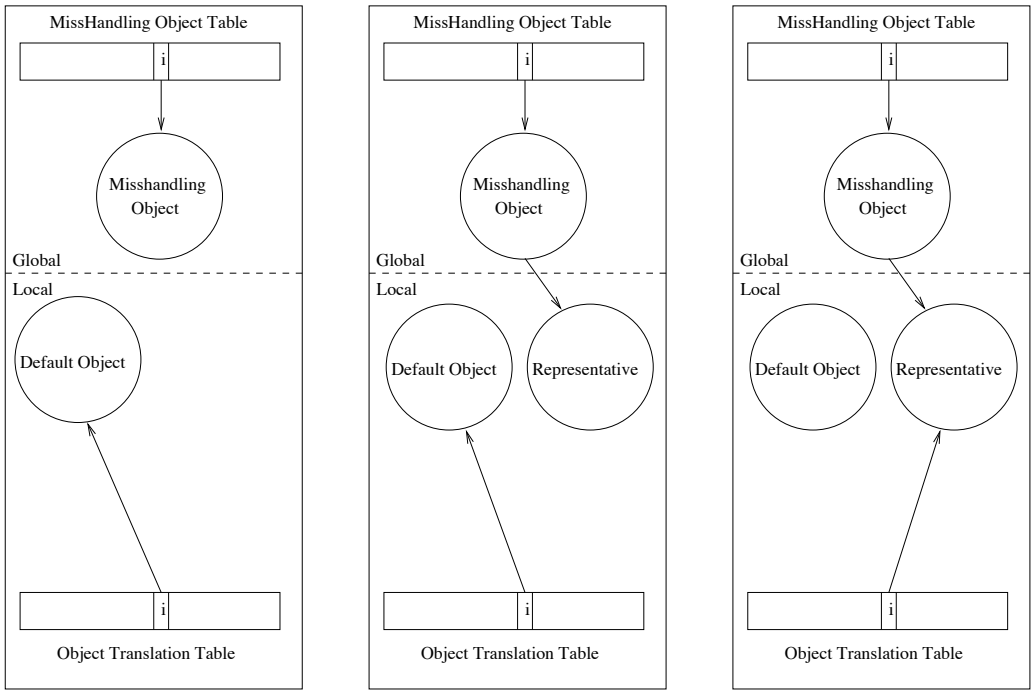


Figure 3.11: Object Translation Table and Misshandling Object Table organization

As stated earlier, the pointer to an entry in the Object Translation Table is called a Clustered Object Identifier. A simple macro performs the dereferences necessary to yield a pointer to the local representative from the Clustered Object Identifier. Clients of a Clustered Object must use the macro on every access to the Clustered Object. The potential for dynamic allocation, deallocation, and migration of representatives, makes it problematic for clients to store direct references to representatives themselves[2].

To avoid excessive resource usage and limit initialization costs, the instantiation of representatives and their assignment to Object Translation Table entries is done lazily in Tornado, in that they are instantiated on first use. To support this, the organization in figure 3.9 is extended with an additional global table called the Misshandling Object Table; see Figure 3.11. Unlike the Object Translation Table, the Misshandling Object Table is global and shared by all processors. For every Clustered Object there is a corresponding entry in the Misshandling Object Table, containing a pointer to the Miss-Handling Object of the corresponding Clustered Object. When a Clustered Object is instantiated, the Miss-Handler of that Clustered Object (which is a regular C++ object) is instantiated, and a pointer to it is installed in the Misshandling Object Table entry for the Clustered Object.

[2]With proper compiler support, the need for an explicit macro to access a Clustered Object could be avoided.



a: Misshandling Object is installed when Clustered Object i is instantiated

b: A reference is made to the Clustered Object i on this processor. The call invokes a method of the Default Object which invokes the handleMiss method of the i Clustered Object's MissHandling Object. This causes the instantiation of a new representative.

c: The default object's method loads the pointer to the new representative returned by handleMiss into the i'th location of the Object Translation Table on this processor.

Figure 3.12: Miss-handling process as seen on one processor

The Object Translation System initializes all Object Translation Table entries to point to a default miss handling object, called the *default* object. The *default* object acts as a trampoline. It directs the first method invocations of a Clustered Object on a processor to the *handleMiss* method of the target Clustered Object's Misshandler by consulting the Misshandling Object Table. The *handleMiss* method then instantiates a new representative if necessary and returning a pointer to the representative responsible for servicing Clustered Object requests on that processor back to the *default* object. The *default* object then replaces the reference to itself in the Object Translation Table entry with the pointer returned by the Misshandling Object. The method call to the Clustered Object is then restarted and proceeds as if the representative were previously installed. Figures 3.12 (a-c) illustrate the miss-handling process. It should be noted, however, that for the miss-handling redirection to work, it is necessary that all externally visible methods of a Clustered Object be implemented as C++ virtual methods by the representatives of the Clustered Object.

The number of Clustered Objects that can exist in an address space is limited by the size of the Object Translation Tables. The goal of Tornado is to support very large Object Translation Tables so that a large number of Clustered Objects may exist. However, large Object Translation Tables can consume considerable real memory. To address this problem, Tornado treats the Object Translation Tables as caches for the current representatives of a processor. The physical pages that store the Object Translation Table entries can be reclaimed when needed. Rather than consuming paging system resources, Tornado expects the Clustered Objects' Misshandling objects to maintain the primary record of which representatives are assigned to which processor.

Consider what happens when an access is made to a Clustered Object whose object translation entry has been reclaimed (i.e. the physical memory on which it should be located is not present). The absence of a physical page, when the corresponding virtual page is accessed, results in a page fault. The page fault handler will subsequently allocate a new physical page. The page is then initialized to contain Object Translation Table entries that point to the default object. The default object will, as before, redirect any call to the MissHandler for the specific Clustered Object. It is the MissHandler's responsibility to recognize, if appropriate, that a representative has already been assigned for this processor and return a reference to it. This is really just a special case of the miss-handling process described earlier. Rather than instantiating a new representative, the MissHandler simply returns a pointer to a previously assigned representative.

As stated earlier, a Clustered Object must obtain a new Clustered Object Identifier for itself when it is created. Similarly, the Clustered Object must destroy itself properly. The de-assignment

function indicates that the Object Translation Table entry can be reused. The Object Translation System requires that every Clustered Object implement a destroy method, which is invoked by a client when it wants to indicate that the Clustered Object is no longer needed. The destroy method invokes a de-assignment function of the Object Translation System and does nothing more. The de-assignment function sets the Object Translation Entries for the target Clustered Object on all processors to a default error object. All methods of the error object return an error status to the invoker. This ensures that any process attempting to access a Clustered Object after it has been de-assigned will receive an error on all method invocations.

Representatives, however, are not deallocated until all processes that may have a temporary reference to the Clustered Object have terminated. When this occurs, the Object Translation system calls a predefined method of the Misshandling Object, called *cleanup*, which deallocates all the representatives.

3.3.2 Requirements on the implementation of Clustered Objects

Tornado's Object Translation System places a number of requirements on the implementation and use of Clustered Objects:

- The external interface of a Clustered Object must be implemented as C++ virtual methods by all representatives.
- Every Clustered Object must provide a Misshandling object that implements:
  - A *handleMiss* method, that is invoked when a miss occurs on a specific processor. This method must return a pointer to the representative that is to be installed in the Object Translation Table entry on the target processor.
  - The instantiation of representatives implementing the clustering strategy chosen for the target Clustered Object.
  - A record of which representatives have been assigned to which processors.
  - A *cleanup* method that is invoked by the Object Translation System to relinquish all resources allocated to the Clustered Object, including those associated with each representative.
- The Clustered Object when created must first instantiate its Misshandling object and return its Clustered Object Identifier to the client. A unique Clustered Object Identifier must be

obtained from the Object Translation system by invoking the Object Translation Table *entry assignment function*.

- The external interface must include a *destroy* method that is invoked by clients to indicate that the Clustered Object is no longer required. This method should invoke the appropriate de-assignment function of the Object Translation System.

- All accesses to a Clustered Object must be made using the given Macro and the Clustered Object Identifier.

Additionally, to maximize locality, the classes should minimize sharing of data.

3.4 Class Representation

This section describes the class representation that was implemented to facilitate the development of Clustered Objects. The class representation serves as a base for the development of Clustered Objects according to the model presented in section 3.2. As stated, the Clustered Object model is based on the Object Translation facilities of Tornado described in the previous section, and the class representation developed is essentially a high-level interface to Tornado's Object Translation System, hiding the Object Translation Systems details.

The Clustered Object model presented in figures 3.6 and 3.7 identifies three separate components:

1. An External Interface.

2. Representatives that implement the External Interface.

3. A Misshandling Object that manages the Representatives and is global to all the Representatives (but internal to the Clustered Object).

These components can be implemented with two C++ classes. One class can be used to define the representatives with the external interface, while the other defines the Misshandling Object. This leads to two class hierarchies from which the two objects of a new Clustered Object can be derived from. Figure 3.13 illustrates the hierarchies that have been implemented. The classes of the two hierarchies provide common default implementations of the methods required by the Clustered Object System. For example, the *ClusteredObject* hierarchy ensures that a *destroy* method is part

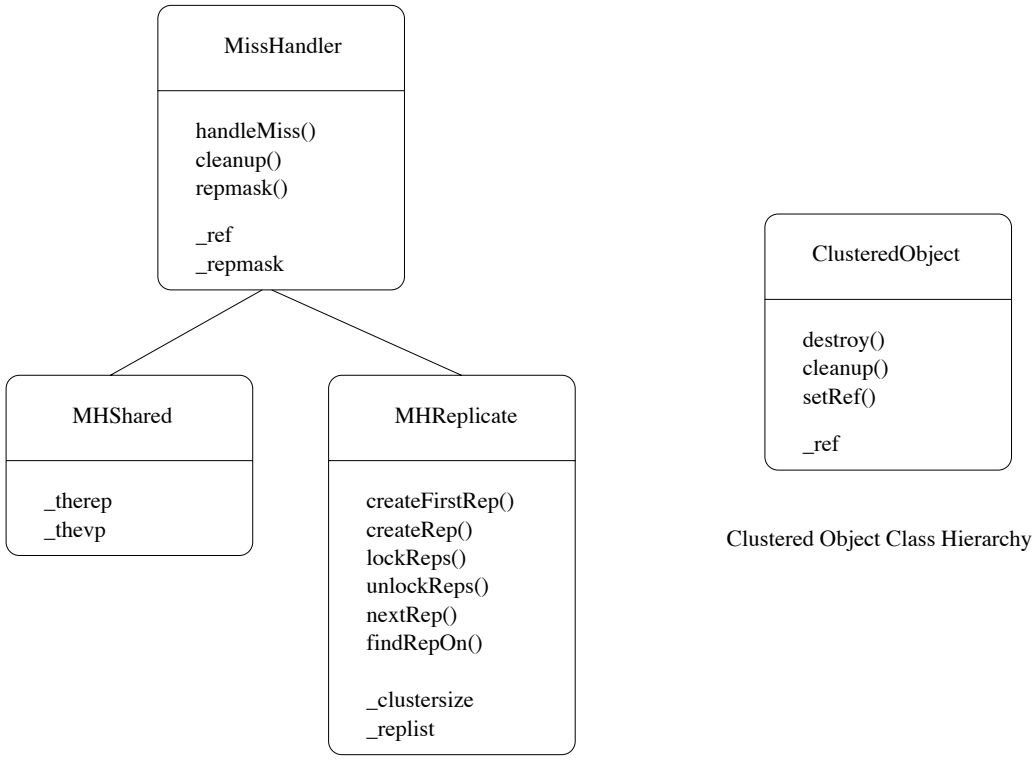


Figure 3.13: Class Hierarchies.

of the external interface for all Clustered Objects. The MissHandling hierarchy defines default *handleMiss* and *cleanup* methods for all MissHandling objects.

The next two subsections discuss the *ClusteredObject* and *MissHandler* hierarchies in more detail. Section 3.4.3 presents the two main representative policies, Shared and Replicated, that are currently supported by the hierarchies. Section 3.4.4 gives two example Clustered Object implementations of an integer counter.

3.4.1 *ClusteredObject* Hierarchy

The *ClusteredObject* class serves two main purposes:

1. To provide the common external interface definitions for all Clustered Objects.

2. To provide the common methods required by all representatives.

When implementing a new Clustered Object, a programmer is expected to define a new class that inherits from *ClusteredObject* or one of its subclasses. We will call this class the representative class. Instances of the representative class are the representatives for the Clustered Object. All external interface methods are defined as public virtual methods of the representative class. Generally, clients of the Clustered Object do not directly instantiate instances of the representative class; the Misshandling object is expected to create, manage and delete all instances of the representative class. In order to standardize Clustered Object instantiation, a programming convention has been adopted, whereby the programmer defines a static *create* method as part of the representative class. This method is responsible for ensuring that an instance of the Misshandling object is created and that the Clustered Object Identifier is passed back to the client.

It is expected that subclasses of *ClusteredObject* will serve as definitions for generic Clustered Object external interfaces and representatives. For example, one might create a subclass called *CounterCO* that defines a generic external interface for counter type Clustered Objects. Similarly, one might create a subclass of *ClusteredObjects* called *BroadCastReps* that implements a generic form of broadcast communication between representatives.

The dual nature of the *ClusteredObject* class hierarchy and the lack of multiple inheritance support in Tornado, makes it difficult to provide arbitrarily combinable, generic external interfaces and generic representative implementations. For example, multiple inheritance would be required to define a Clustered Object that supports both the generic counter interface defined by *CounterCO* and the generic broadcast communications capabilities defined by *BroadCastReps*. An alternative

approach might be to create a new class that explicitly implements both the external interface of *CounterCO* and the behaviour of *BroadCastReps*. This method however, requires re-implementing the code of at least one of the pre-existing classes within the new class. This leads to added complexity in the class hierarchy and potential for errors.

3.4.2 *MissHandler* Hierarchy

The classes of the *MissHandler* hierarchy are the base classes from which the Misshandling object of a Clustered Object is derived. These classes serve two purposes:

1. They define and implement the methods that the Object Translation System requires of the MissHandler.

2. They define and implement the representative management policies for a Clustered Object.

When implementing a new Clustered Object, the programmer must ensure that a Misshandling object is instantiated to manage its representatives. The subclasses of the *MissHandler* class implement different representative management policies. Shared and replicated representative policies are implemented by the *MHShared* and *MHReplicated* classes, respectively (these polices are described in the following subsection). A programmer is free to add additional policies or to provide specializations of the current ones through standard inheritance.

3.4.3 Shared and Replicated Clustered Objects

The three data management policies that we will focus on are sharing, replication and partitioning. To support these policies, the class representation must be able to support two distributions of representatives; shared and replicated.

In the shared case, only one shared representative is needed. The single representative contains the one and only copy of the data for the entire Clustered Object. Any processor that attempts to access the Clustered Object, is always directed to this one representative. The miss-handling behaviour used to implement a shared representative Clustered Object is simple: after instantiating the shared representative, the Misshandling object simply needs to return a pointer to it on every translation miss. The *MHShared* class implements this behaviour. The *MHShared* class can be directly instantiated to produce a MissHandling object for any shared representative Clustered Object. This allows an implementor to define a shared representative Clustered Object without having to explicitly define a new Misshandling class.

A Clustered Object that uses *MHShared*, displays the same behaviour as a standard C++ object. Although this seems like a trivial use of the Clustered Object Model, it is important in that the shared data policy is often required to efficiently implement frequently read and written shared data, and in that it becomes trivial to turn any standard C++ class into a shared representative Clustered Object. This helps encourage incremental design and optimization of Clustered Object based systems. An implementor can first use a naive shared implementation of a data structure and then later return and selectively replace such implementations with optimized versions as needed.

The replicated and partitioned data policies require multiple representatives. A given instance of the representative class can either store a replica or a partition of the Clustered Object's data. How the instance is used is completely dependent on the methods of the representatives. For example, in the case of a partitioned counter, each representative might have an integer value that is treated as only a portion of the total value. On the other hand, in the case of a replicated identifier, each representative's integer value could be treated as a local replica of the identifier's value.

MHReplicate implements the necessary miss-handling behaviour to support multiple representatives. It also supports arbitrary fixed clustering degrees. Unlike the *MHShared* class, members of the *MHReplicate* class cannot be instantiated directly. It is first necessary to define a subclass of *MHReplicate* that meets the needs of the Clustered Object. In particular, it must implement the *createFirstRep* and *createRep* methods, which together define a Clustered Object's representative instantiation behaviour. *CreateFirstRep* defines the instantiation of the first representative, while *createRep* defines the instantiation of all other representatives.

The *MHReplicate* class automatically records references to all representatives and the processors to which they have been assigned. It ensures that *createFirstRep* or *createRep* is invoked only once per cluster. The clustering degree can be passed as an initialization argument to a subclass of *MHReplicate*. The *MHReplicate* class also ensures correct destruction of the representatives. The Miss Handling class *MHReplicate* provides the following convenience functions that will be referred to as the Miss-Handling *representative management functions*: *lockReps*, *unlockReps*, *nextRep*, and *FindRepOn*. These functions allow access to the set of representatives maintained by the Miss-Handling object. A representative can use these functions to lock, unlock and iterate over the set of representatives that currently compose the Clustered object[3]. The *FindRepOn* method returns a pointer to the current representatives associated with a specified processor. It returns a null value

[3]Locking the set of representatives ensures that no changes, additions or deletions, to the set will occur.

```
class integerCounter : public ClusteredObject {
public:
    virtual void value(int &val)=0;
    virtual void increment()    =0;
    virtual void decrement()    =0;
    virtual      ~integerCounter(){}
};

typedef integerCounter **integerCounterRef;

class SharedCounterCO : public integerCounter {
    int _count;
    MHShared _mh;
    SharedCounterCO() : _mh(this) { _count=0; }
    integerCounterRef ref() { return (integerCounterRef)_ref; }
public:
    static integerCounterRef create() {
        return (integerCounterRef)((new SharedCounterCO()->ref()));
    }
    virtual void value(int &val)  { val=_count; return; }
    virtual void increment()      { FetchAndAdd(&_count,1); }
    virtual void decrement()      { FetchAndAdd(&_count,-1); }
};
```

Figure 3.14: C++ Code that implements the Shared Counter as a Clustered Object

if no representative is currently assigned.

3.4.4 Examples

This subsection presents two Clustered Object implementations of the integer counter: *SharedCounterCO* and *CounterLocalizedCO*, using a shared representative and multiple representatives, respectively.

Shared Representative Example

Figure 3.14 shows an example of how to implement the shared counter as a Clustered Object. The following list summarizes the differences between the Clustered Object version in figure 3.14 to the non-Clustered Object version presented in figure 2.4 on page 17:

- The *integerCounter* class is now inherited from the *ClusteredObject* class. All classes which now implement the *integerCounter* interface will be Clustered Object representative classes.
- The *SharedCounterCO* has been given a member of *MHShared* (*_mh*) that serves to define a MissHandler for the Clustered Object.

- The constructor of *SharedCounterCO* has been made private and an initializer for *_mh* has been added to the constructor.
- A static create method has been added to *SharedCounterCO* that creates a new instance of the Clustered Object and returns the Clustered Object Identifier to the instantiator.

By inheriting from the *ClusteredObject* class, the *SharedCounterCO* class inherits the default *destroy* and *cleanup* methods to facilitate correct Clustered Object destruction. The *ClusteredObject* class also ensures that the representative has a copy of the Clustered Object Identifier for the Clustered Object it belongs to.

The instantiation of the representative will also ensure the instantiation of the Misshandling object, as it is embedded in the representative.

To avoid client code from trying to directly instantiate instances of the *SharedCounterCO* class, the constructor for the class has been made private. C++ access rules will then restrict the instantiation of the *SharedCounterCO* class to the member methods of the class itself. Instead, a static *create* method has been added to allow clients to create instances of the Clustered Object. This method creates an instance of the *SharedCounterCO* class. It instantiates both the sole representative and the MissHandling object embedded in it. The *create* method also passes the Clustered Object Identifier back to the creator.

The use of a static *create* method and the privatization of the default constructor, are programming conventions for Clustered Objects that the Programmer needs to be aware of. Such conventions could be avoided if explicit compiler support for Clustered Objects were available.

Replicated Representative Example

Figure 3.15 shows the implementation of a clustered counter using a local representative on each processor that maintains a local count value on the processor it is assigned to. This example is similar to the non-Clustered Object example presented in figure 2.8 on page 20.

Perhaps the most noticeable feature of the code in figure 3.15 is the addition of a separate class definition for the MissHandling Object. The *CounterLocalizedCOMH* class is a subclass of *MHReplicate* and defines how representatives are to be instantiated. This done by providing implementations for the *createFirstRep* and *createRep* methods. *CreateFirstRep* and *createRep* are essentially the same (figure 3.15) since all representatives must be instantiated identically. This would not be the case, for example, if one required the ability to specify the starting value of the

```
class CounterLocalizedCO : public integerCounter {
    class CounterLocalizedCOMH : public MHReplicate {
    public:
        virtual ClusteredObject * createFirstRep() {
            return (ClusteredObject *)new CounterLocalizedCO;
        }
        virtual ClusteredObject * createRep() {
            return (ClusteredObject *)new CounterLocalizedCO;
        }
    };
    friend class CounterLocalizedCO::CounterLocalizedCOMH;
    struct counter {
        int val;
        char pad[SCACHELINESIZE - sizeof(int)];
    } _count;
    CounterLocalizedCO() { _count.val=0; }
public:
    static integerCounterRef create() {
        return (integerCounterRef)((new CounterLocalizedCOMH())->ref());
    }
    virtual void value(int &val) {
        MHReplicate *mymh=(MHReplicate *)MYMHO;
        CounterLocalizedCO *rep=0;
        val=0;
        mymh->lockReps();
        for (void *curr=mymh->nextRep(0,(ClusteredObject *&)rep);
             curr; curr=mymh->nextRep(curr,(ClusteredObject *&)rep))
           val+=rep->_count.val;
        mymh->unlockReps();
    }
    virtual void increment() { FetchAndAdd(&(_count.val),1); }
    virtual void decrement() { FetchAndAdd(&(_count.val),-1); }
};
```

Figure 3.15: C++ Code for the LocalizedCounter Clustered Object

counter.

In the case of a Clustered Object with multiple representatives, the MissHandling Object cannot be directly embedded into the representatives. The MissHandling Object must be created at the time the Clustered Object is created. As a result the static *create* method of *CounterLocalizedCO* creates only an instance of the *CounterLocalizedCOMH*. The miss-handling behaviour it implements will ensure that representatives are created as needed.

It is interesting to compare the methods of the Clustered Object *CounterLocalizedCO* class with the methods of the non-Clustered Object *CounterLocalized* class. In the non-Clustered Object variant, each method must explicitly locate the local value within the array of all local values, while the Clustered Object variant does not need to identify local data values, since the data members of the representatives are the local values. The non-Clustered Object variant also needed to construct all the local counter values at initialization time by remotely invoking an initialization procedure on every processor. In contrast, the Clustered Object variant avoids this complexity, and representatives are instantiated on first use. The Clustered Object variant dynamically adjusts to the number of processors that access it.

The *value* method of the non-Clustered Object version uses the global array of local count values to calculate the total value of the counter. The Clustered Object version makes use of the Miss-Handler representative management functions to identify the representatives so that the total value can be calculated. The *MYMHO* utility macro provided by the *ClusteredObject* class is used by the representative to locate its MissHandling Object. Since the Clustered Object model allows for dynamic instantiation of representatives, it is necessary to lock and unlock the list of representatives prior to and after traversal of the list. Once the lock is obtained, an initial call to *nextRep* returns the first representative in the list. Additional calls return the successive representatives.

3.4.5 Summary

The class representation we developed, provides a base for implementing Clustered Objects according to the Clustered Object model supported by Tornado's Object Translation system. The two main types of objects that compose a Clustered Object are (i) representatives and (ii) Misshandling objects. There is one Misshandling object per Clustered Object and the Misshandling object is responsible for managing one or more representatives.

The representatives define and implement the external interface for the Clustered Object. The representatives of a Clustered Object are built by implementing a subclass of the *ClusteredObject*

class. By convention, an implementor of a Clustered Object must provide a static C++ *create* method that creates instances of the Clustered Object.

We have implemented two separate types of MissHandling objects: *MHShared* and *MHReplicate*. *MHShared* can be used to create Clustered Objects that only support a shared representative policy, where all requests to a Clustered Object are directed to a single shared representative. The *MHReplicate* class is a base to build more general misshandling objects that support replicated representative Clustered Objects. Subclasses of *MHReplicate* instantiate multiple local representatives. The local representatives can be implemented to support replication or partitioning of the Clustered Object's data. The *MHReplicate* class supports fixed clustering degrees by ensuring that only one representative is instantiated per cluster of processors. The number of processors in a cluster can be specified as an initialization parameter to the MissHandling object.

Chapter 4

Examples of Clustered Object implementations

Clustered Objects facilitate the implementation of traditional objects as a collection of representatives on an SMP. When implementing a given object there are many options as to where and how data can be located, managed and accessed. Some data can be local to a representative, while other data can be global across all representatives. Similarly, some data can be accessed via shared memory, while other data are accessed via remote procedure calls.

This chapter attempts to illustrate some of these options through example Clustered Object implementations. The first section will present additional Clustered Object implementations of the Counter data structure discussed in the previous chapters. The next section will present a more complex SMP data structure from the literature and three Clustered Object implementations of it.

4.1 Counters

The *CounterLocalizedCO* Clustered Object in figure 3.15 (page 45) uses the Miss-Handling representative management functions to implement its *value* method. Although the MissHandler representative management functions are convenient to use, they are not optimized for any specific Clustered Object type. Hence, an implementor may choose to implement and coordinate sharing between representatives by other, more efficient means. These can include: providing explicit representative organizations, use of representative global data, and use of function shipping. The following three subsections present examples that illustrate these methods.

4.1.1 Explicit Representative Organization

By using specific knowledge about the object being implemented and its use, the implementor can explicitly organize the representatives to better support global operations. For example, the representatives of the counter Clustered Object could be linked together to form a circular chain. This allows the global summation function to be naturally implemented as a traversal of the chain. Although this appears similar to the use of the Miss-Handling representative management function *nextRep*, it differs in how the chain is implemented. In the case of the Miss Handler representative management functions, the chain is implemented as a separate linked list maintained by the Miss Handler, with each node containing a pointer to a specific representative. Explicitly organizing the representatives into a chain avoids the need to access the Miss Handler.

To implement the counter as a chain, each representative must maintain a pointer to the next representative. It is necessary to ensure that modifications (insertions and deletions) to the list and the global sum operation (which traverses the list) are properly synchronized. Traditionally this is achieved with proper locking. In this particular case, however, locking can be avoided by observing that:

1. Deletions from the representative chain do not conflict with summation operations because representatives, once created, exist for the remaining life-time of the Clustered Object.

2. Inserting a representative during a summation does not affect the accuracy of the sum. As already observed, the *value* method is not atomic with respect to updates and as such its result is approximate. If a representative is inserted during a summation, its value may or may not be included in the sum. This is no different than increments or decrements that occur during the summation operation. Insertions thus need not be made atomic with respect to the summation.

3. To maintain the consistency of the pointers that form the chain, it is possible to carefully order the sequence in which pointers in the chain are modified and thus prevent dereferencing of dangling pointers.

Figure 4.1 presents a Clustered Object that is implemented as a chain of representatives. Each representative has a pointer to the next representative in the chain. The *createFirstRep* and *createRep* methods maintain the chain of representatives by correctly linking in new representatives when they are created. Finally, the *value* method now simply sums the values from each representative

```
class CounterLinkedCO : public integerCounter {
    class CounterLinkedCOMH : public MHReplicate {
        CounterLinkedCO *_first,*_last;
    public:
        virtual ClusteredObject * createFirstRep() {
            _first=_last=new CounterLinkedCO;
            _last->_next=_first;
            return _last;
        }
        virtual ClusteredObject * createRep() {
            CounterLinkedCO *tmp=new CounterLinkedCO();
            tmp->_next=_first; _last->_next=tmp; _last=tmp;
            return _last;
        }
    };
    friend class CounterLinkedCO::CounterLinkedCOMH;

    int _count;
    CounterLinkedCO *_next;
    char pad[SCACHELINESIZE - sizeof(int) - sizeof(CounterLinkedCO *)];

    CounterLinkedCO() { _count=0; _next=0; }
public:
    static integerCounterRef create() {
        return (integerCounterRef)((new CounterLinkedCOMH())->ref());
    }
    virtual void value(int &val) {
        val=_count;
        for (CounterLinkedCO *p=_next;
             p!=this; p=p->_next)
            val+=p->_count;
    }
    virtual void increment() { FetchAndAdd(&_count,1); }
    virtual void decrement() { FetchAndAdd(&_count,-1); }
};
```

Figure 4.1: C++ CounterLinkedCO implementation.

as it traverses the chain.

Many other organizations such as single linked lists, doubly linked lists and trees may be useful, depending on the type of Clustered Object being implemented.

4.1.2 Representative Global Shared Data

By making use of shared memory, an implementor is free to allocate data that can be accessed globally by all the representatives of a Clustered Object. For example, rather than adding links to each representative, as in the previous example, a shared array of representative pointers could be used. A pointer to each representative would be recorded in the array and the summation procedure would make use of the array to visit each representative. The implementor must correctly manage and maintain the global data. She must make sure the global data is allocated and deallocated correctly, that each representative is given access to it and that its consistency is maintained.

When implementing a counter that makes use of a shared array of representative pointers it seems natural to have the Miss Handler allocate and deallocate the array when the Clustered Object is allocated and deallocated, respectively. It can ensure that each representative has access to the array by passing the array pointer to each representative when it is constructed.

The array must be large enough to hold a pointer to each representative that might be instantiated. This can be achieved by allocating an array with one element per processor, so that it can be safely indexed by processor number. Assigning each processor its own element within the array avoids the need for synchronization when recording a representative in the array. Remembering the scenario presented in Chapter 2, we expect that the counter will be accessed on every processor, thus the array will be completely utilized.

Similar to the example in the previous subsection, the approximate nature of the counter makes it unnecessary to ensure that updates to the array are atomic with respect to traversal of the array by the *value* method. However, it is necessary to ensure that the *value* method does not attempt to access a representative that does not exist. This is easily ensured by having all elements of the array initialized to null and having the *value* method check the validity of each element prior to dereferencing it.

Figure 4.2 presents an implementation using a shared array. The Miss Handler's *createFirstRep* method creates the shared array and passes to the first representative a reference to this array. The *createRep* method passes the reference to this array to all other representatives as they are instantiated. Each representative records a reference to itself in the array when it is instantiated.

```
class CounterArrayCO : public integerCounter {
    class CounterArrayCOMH : public MHReplicate {
        CounterArrayCO **reparray;
    public:
        virtual ClusteredObject * createFirstRep() {
            reparray=new CounterArrayCO *[NUMPROC];
            for (int i=0;i<NUMPROC;i++) reparray[i]=0;
            return new CounterArrayCO(reparray);
        }
        virtual ClusteredObject * createRep() {
            return new CounterArrayCO(reparray);
        }
        virtual ~CounterArrayCOMH() {
            delete[] reparray;
        }

    };
    friend class CounterArrayCO::CounterArrayCOMH;

    int _count;
    char pad[SCACHELINESIZE - sizeof(int)];
    CounterArrayCO **_reps;

    CounterArrayCO(CounterArrayCO **reparray)
    {
        _count=0;
        _reps=reparray;
        reparray[MYVP]=this;
    }

public:
    static integerCounterRef create() {
        return (integerCounterRef)((new CounterArrayCOMH())->ref());
    }
    virtual void value(int &val) {
        val=0;
        for (int i=0;i<NUMPROC;i++)
            if (_reps[i]) val+=_reps[i]->_count;
    }
    virtual void increment() { FetchAndAdd(&_count,1); }
    virtual void decrement() { FetchAndAdd(&_count,-1); }
};
```

Figure 4.2: CounterArrayCO implementation.

```
class ArrayofRepPointers;
typedef ArrayofRepPointers **ArrayofRepPointersRef;

class ArrayofRepPointers : public ClusteredObject {
    int _size;
    ClusteredObject **_reps;
    MHShared _mh;

    ArrayofRepPointers(int &size) : _mh(this) {
        _size=size;
        _reps=new ClusteredObject *[_size];
        for (int i=0;i<_size;i++) _reps[i]=0;
    }
    ArrayofRepPointersRef ref() { return (ArrayofRepPointersRef) _ref; }
public:
    static ArrayofRepPointersRef create(int &size) {
        return (new ArrayofRepPointers(size))->ref();
    }
    virtual void getValueAt(int &index, ClusteredObject* &value) {
        if (index>_size) value=0;
        else value=_reps[index];
    }
    virtual void setValueAt(int &index, ClusteredObject* &value) {
        if (index<_size) _reps[index]=value;
    }
};
```

Figure 4.3: A Clustered Object that implements a shared Array of representative pointers.

The Miss Handler also has a destructor that deallocates the array when the Clustered Object is destroyed. The *value* method iterates across the array, accessing the count values of all the representatives.

A natural extension to the above example is to use a Clustered Object to implement the global array. Figures 4.4 and 4.3 shows how this might be done. The Miss Handler for the counter instantiates an instance of the new array Clustered Object and passes to each representative of the counter, as it is instantiated, a reference to the array Clustered Object.

While, the shared memory of a SMP provides an easy way to implement representative global data within a Clustered Object, using a separate Clustered Object to implement global data can provide the implementor with greater flexibility. For instance, the shared array in the above example might later be replaced with a partitioned array, without affecting the implementation of the *CounterArrayCOCO* in any way. This allows an implementor to customize the counter for different access patterns by simply replacing one standard component with another more suited one. However, there is a space and time overhead associated with the use of a new Clustered Object.

```
class CounterArrayCOCO : public integerCounter {
    class CounterArrayCOCOMH : public MHReplicate {
        ArrayofRepPointersRef _reparray;
    public:
        virtual ClusteredObject * createFirstRep() {
            int procs=NUMPROC;
            _reparray=ArrayofRepPointers::create(procs);
            return new CounterArrayCOCO(_reparray);
        }
        virtual ClusteredObject * createRep() {
            return new CounterArrayCOCO(_reparray);
        }
        virtual ~CounterArrayCOCOMH() {
            DREF(_reparray)->destroy();
        }
    };
    friend class CounterArrayCOCO::CounterArrayCOCOMH;

    int _count;
    char pad[SCACHELINESIZE - sizeof(int)];
    ArrayofRepPointersRef _reps;

    CounterArrayCOCO(ArrayofRepPointersRef reparray)
    {
        _count=0;
        _reps=reparray;
        ClusteredObject *me=this;
        DREF(_reps)->setValueAt(MYVP,me);
    }
public:
    static integerCounterRef create() {
        return (integerCounterRef)((new CounterArrayCOCOMH())->ref());
    }
    virtual void value(int &val) {
        ClusteredObject *rep;
        val=0;
        for (int i=0;i<NUMPROC;i++)
        {
            DREF(_reps)->getValueAt(i,rep);
            if (rep) val+=((CounterArrayCOCO *)rep)->_count;
        }
    }
    virtual void increment() { FetchAndAdd(&_count,1); }
    virtual void decrement() { FetchAndAdd(&_count,-1); }
};
```

Figure 4.4: C++ CounterArrayCO implementation.

Using a separate Clustered Object means using additional Object Translation System resources and per-Clustered Object memory overhead[1]. Accesses to the array will also suffer a slight increase in overhead due to the double de-reference required to access the Clustered Object[2].

4.1.3 Function shipping

So far, all the examples have used shared memory to access the individual counters of the representative when calculating the global sum. All the data is thus brought to the processor on which the global sum is calculated. This model of computation is often referred to as data shipping. Function shipping, instead, moves the computation to the processors on which the data resides. For example, the *value* method of a counter can be implemented using remote procedure calls. The global value of the counter could be obtained by remotely invoking a local *sum* method on each processor successively. The local *sum* method would take a value as an argument and add to it its current local count value, returning the sum.

Remote procedure calls are not likely to make performance sense for the implementation of a Counter, but we describe it here nevertheless for example purposes. In some cases, function shipping can be cost effective. For example, function shipping can often eliminate the need for locking, as data accesses can be forced to occur on one processor only. Also, function shipping can reduce false sharing and coherency traffic in general, as multiple cached copies of data are avoided. On the other hand, remote procedure calls can be expensive. They are often implemented using the cross-processor interrupt facility, with both a direct overhead for the interrupt handling and indirect overhead due to instruction cache disturbance on the remote processor. The implementor must decide if function shipping is appropriate based on the expected overheads. If remote operation is complex and invoked relatively infrequently, then its overhead, when amortized over the total number of accesses to the Clustered Object, may prove to be smaller than the overhead of data shipping. Of course, a Clustered Object is also free to mix the use of data and function shipping.

Figure 4.5 shows a counter implementation using function shipping. A *sum* method has been added to the representatives, which adds the representatives' current count value to the value passed in. The *value* method makes use of Tornado's remote procedure call facilities to successively invoke the *sum* method on each processor. The *value* method need not be concerned with the possibility that a representative does not exist on a given processor, as the Clustered Object system will ensure

[1] In our implementation, the memory costs for a Clustered Object that uses an *MHReplicate* Misshandler are: 2 primary cache lines for the Misshandler and 1 primary cache line per representative, where each cache line is 32 bytes.

[2] On a modern processor, this cost is negligible if the pointers' values are cached.

```
class CounterRemoteCO : public integerCounter {
    class CounterRemoteCOMH : public MHReplicate {
    public:
        virtual ClusteredObject * createFirstRep() {
            return (ClusteredObject *)new CounterRemoteCO;
        }
        virtual ClusteredObject * createRep() {
            return (ClusteredObject *)new CounterRemoteCO;
        }
    };
    friend class CounterRemoteCO::CounterRemoteCOMH;
    struct counter {
        int val;
        char pad[SCACHELINESIZE - sizeof(int)];
    } _count;
    CounterRemoteCO() { _count.val=0; }

    virtual TornStatus sum(int *val) { *val+=_count.val; return 0; }

public:

    static integerCounterRef create() {
        return (integerCounterRef)((new CounterRemoteCOMH())->ref());
    }
    virtual void value(int &val) {
        int *res=new int;
        *res=_count.val;
        for (int i=0;i<NUMPROC;i++)
            if (i!=MYVP) {
                RFUNC1(i,(CounterRemoteCO **)_ref,
                       CounterRemoteCO::sum,res);

            }
        val=*res;
        delete res;
    }
    virtual void increment() { FetchAndAdd(&(_count.val),1); }
    virtual void decrement() { FetchAndAdd(&(_count.val),-1); }
};
```

Figure 4.5: CounterRemoteCO implementation

creation of a representative to satisfy the request if one does not already exist. This does not impose any extra overhead based on the scenario of chapter 2 for which the counter is to be used, as it is expected that all the processors will eventually have its own representative. In general, however, a different implementation may query the Miss Handler representative management functions to determine which processors to direct the remote procedure invocations to.

4.1.4 Clustering Degree

The previous subsections focused on how to implement the global behaviour of a Clustered Object. This section will consider the clustering degree of an object. In the previous examples, it was assumed that each processor would have its own representative, and hence the clustering degree was assumed to be 1. This need not be the case, as a representative can be shared between some subset of processors. For example, a Counter Clustered Object might be defined to have a clustering degree of 4, in which case clusters of four processors would share a representative.

To simplify the specification of arbitrary clustering degrees, support has been put into the *MHReplicated* Miss Handling class. When constructing a Miss Handler of type *MHReplicate*, a clustering degree can be specified as an argument. The class will ensure that *createRep* will be invoked only when a new representative is necessary; if a miss occurs on a processor that belongs to a cluster for which a representative already exists, then the *handlemiss* function of the *MHReplicate* class will install a pointer to the existing representative into the Object Translation Table rather than instantiating a new representative.

It is up to the implementor to ensure that the implementation can support various degrees of clustering. For example, *CounterLinkedCO* would function correctly at different clustering degrees without modification, whereas the *CounterRemoteCO* of the previous subsection would not, as it, by invoking the sum function on each processor, would sum each representatives' value 4 times (if the clustering degree is 4).

The *CounterArrayCO* and *CounterArrayCOCO* clustered objects would also function properly without modification for different degrees of clustering. However, in both cases, if the clustering degree is 4 then 3/4 of the array would be unused, since the array of representative pointers is allocated to be equal to the number of processors. This could be solved by modifying the allocation of the array to be 1/4 the size and to ensure that all indexing into the array is modified to use a cluster number rather than processor number.

The clustering degree that is appropriate for a given instance of a clustered object depends on

its usage and will typically need to be experimentally determined. Thus, a Clustered Object should be designed to function at arbitrary clustering degrees, so the implementor may wish to make the clustering degree an instantiation parameter.

4.2 Software Set-Associative Cache

This section examines how a more sophisticated object can be implemented as a clustered object. The object being considered is a software set-associative cache that has been of use in multiprocessor operating systems. Peacock et al. developed Software Set-Associative Caches (SSAC) as a general software cache architecture to address contention problems that occur with the naive software caches often being used [28],[29].

This section will focus on a straight-forward SSAC architecture, as is described in [29] and [28]. The structure of SSAC is similar to that of a hardware cache. Specifically, the simple SSAC implements a write back cache with fixed set-associativity and a least recently used replacement policy within a set. Unlike a typical hardware cache, however, the SSAC is designed to be shared by multiple processors.

Figure 4.6 illustrates the basic shared SSAC structure and how it is used to cache objects. Every cache object (data item) to be cached by the SSAC must have a unique Cache Object Identifier (COID). The complete range of COIDs form the COID space, represented by the rectangular array at the top of the figure. Processors, represented by circles, wishing to make a request for data associated with a given COID, must apply a Hash Function (HF) to the COID. The Hash Function maps a COID onto a unique hash queue via a Queue Index (QI). The shared array of hash queues, located in the middle of the diagram, forms the main body of the SSAC. Each hash queue is implemented as a fixed size array of hash entries (the size of this array is the associativity of the SSAC). For clarity, only one of the hash entry arrays is illustrated on the left-most hash queue. Each hash entry is capable of storing one cached object along with some state flags associated with the data. A blow-up of one of the hash entries is shown in the bottom center of the figure.

Each hash queue has a lock that protects all the entries on that queue. To allow for a least recently used eviction policy, a counter within each hash queue and last-used values within each hash entry are maintained. On each access to a hash queue, the hash queue's counter is updated. This counter serves as a global access count for the given hash queue. On each access to a hash entry, the value of the counter of the hash queue to which the entry belongs is recorded in the

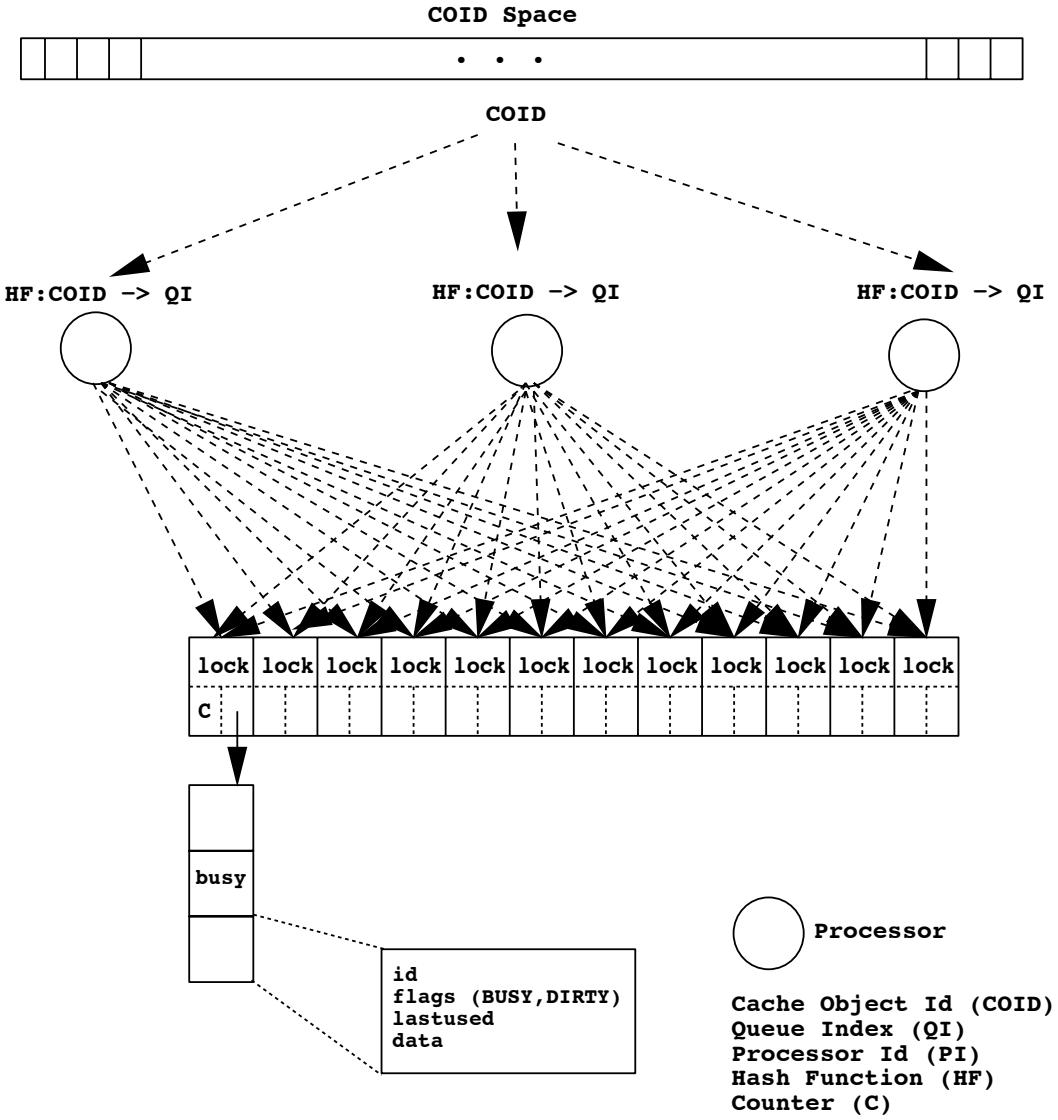


Figure 4.6: Simple Shared SSAC

last-used field of the entry. The entry with the smallest last-used value, is the least recently used entry with respect to the other entries in the same queue.

The two main operations on an SSAC are *get* and *put*. The *get* operation takes a COID as input and returns a pointer to a hash entry that contains the corresponding data. Abstractly, it accomplishes this by using the COID to index into the hash table. If the target cache object is found, then a pointer to the appropriate hash entry is returned. If it was not found, then the least recently used hash entry on the hash queue is identified. If that entry is dirty, it is written back to a backing store, and the target cache object is loaded from the backing store into the least recently used hash entry, before a pointer to it is returned.

A hash entry can be marked as busy by setting a flag appropriately. In figure 4.6, the middle entry of the array of hash entries pointed to by the left-most hash queue is marked busy. *Get* will spin until its target entry is no longer busy, allowing synchronized access to the data. For this study, we extended the *get* operation to implement a multiple readers/single writer policy[3] and thus differentiate between *get_for_read* and *get_for_write*. In the case of a *get_for_read*, the entry will not be marked busy. However, in the case of *get_for_write*, the entry will be marked busy. In either case, if an entry is identified as the target of the *get* and it is marked busy, then both *get* operations will spin on the busy flag of the target entry until it is unset. The busy flag is unset as a side-effect when the *put* operation completes.

The SSAC described above will be referred to as the Simple Array SSAC. The following three subsections will explore three different implementations of the Simple Array SSAC Architecture in the clustered object framework:

1. Shared SSAC
2. Replicated SSAC
3. Partitioned SSAC

4.2.1 Shared SSAC

Perhaps the most straight-forward SSAC realization as a clustered object would be to directly implement the Simple Array SSAC with one shared representative object and two visible operations: *get* and *put*. The *get_for_read* and *get_for_write* operations can be implemented as one *get* operation

[3]In our implementation a writer is free to proceed regardless of readers. It is assumed that a reader makes a copy of the data and that they will verify the validity of the data prior to using it.

that takes a parameter indicating *read* or *write*. By extending code fragments presented in [29] and applying standard object-oriented techniques, one can arrive at a straight-forward set of C++ classes that implement the Simple Array SSAC. The Simple Array SSAC is implemented as two classes: *SSAC* is an abstract class that defines the interface of the SSAC classes, and *SSACSimpleSharedArray* inherits and implements the SSAC interface. By separating the interface from the implementation, it is easier to provide different implementations at a later point without needing to modify client code.

The *SSACSimpleSharedArray* class implements the structures of figure 4.6. It contains an array of hash queue structures and each hash queue structure contains a lock, a counter and a pointer to an array of cache entries. Each cache entry contains a COID, a flags field and a last-used field. Initially, the pointers to the array of cache entries in the hash queue array elements are set to null. The space required for a given array of entries is allocated on first access to the given hash queue. The SSACSimpleSharedArray is parameterized so that the number of hash queues and the number of elements in the hash queues can be set at creation time.

The standard C++ *SSACSimpleSharedArray* class, as described so far, can be converted into a shared clustered object with little programming effort. The following are the steps required:

1. Make the *SSAC* class a sub class of *ClusteredObject*

2. Add a *MHShared* member to the *SSACSimpleSharedArray* class and a corresponding initializer to *SSACSimpleSharedArray*'s constructor initialization list.

3. Add a public static create method to the *SSACSimpleSharedArray* class and hide its current constructor by making it private.

The clustered object framework requires that all externally visible methods of the clustered object are virtual, but in this case since the interface for *SSACSimpleSharedArray* has already been defined as a set of virtual functions in the abstract base class, the requirement of having all external methods virtual is already met. In total, the conversion amounts to the modification of 3 lines of code and the addition of 6 new lines of code to define the *create* method.

A user of the *SSACSimpleSharedArray* would call the static create method to instantiate a new instance of the *SSACSimpleSharedArray* Clustered Object. The Clustered Object reference returned by the *create* method is used to access the instance. This Clustered Object SSAC implementation will share one representative for all processors and will be referred to as the Shared SSAC implementation.

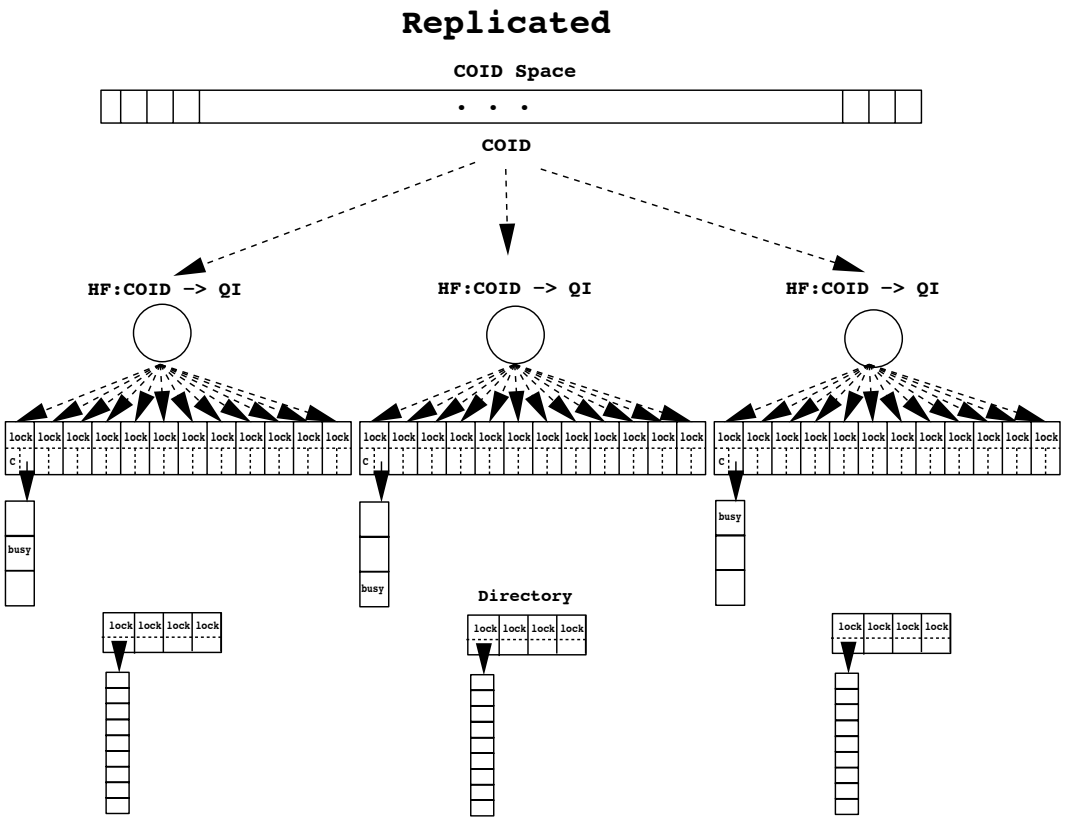


Figure 4.7: Replicated SSAC

In the Simple Shared SSAC, all processors compete for access to the same shared hash table. The Simple Shared SSAC structure has already been tuned for concurrency, as it was found by Peacock et al. that having individual locks on each hash queue leads to minimal lock contention [29]. However, the Simple Shared SSAC does nothing to improve cache or data locality.

4.2.2 Replicated SSAC

This subsection explores an implementation of the SSAC architecture that uses multiple representatives to replicate the Simple Array SSAC. We refer to this implementation as the Replicated SSAC. Figure 4.7 illustrates such an organization. It replicates the Shared SSAC structure on a per-processor basis. Consistency between the replicas is maintained using a directory-based write-update cache protocol [34, 20]. It was felt that better performance may actually be achieved by using an invalidate protocol but that the added complexity of the update protocol would better explore the expressiveness of the Clustered Object approach. From the clients' perspective, the Replicated SSAC and the Shared SSAC should appear to be identical and thus provide the same interface and functionality, even if the performance of the two may differ depending on the access

pattern.

Although the interface remains the same, the implementation of *get* and *put* must be modified substantially to maintain coherency between the replicas. Analogous to hardware caches, it is unnecessary to keep the replicas identical in what they contain. However, if copies of the same element exist in two replicas then the copies need to be kept coherent. A directory can be used to keep track of the number and location of the copies.

A *get* operation is always directed to a local representative of the SSAC. If the requested object is not found in the local replica, then the directory is consulted to see if the object exists in any of the other replicas. If it does, the data is copied to the local replica and the directory is updated to reflect the new copy. If the value does not exist in any of the replicas, then the object must be loaded from backing store, as in the Shared SSAC case, and the directory must be updated accordingly. To synchronize data access, *get_for_write* must now ensure that the busy bits on all of the copies are set, and *put* must update all copies and unset their busy bits. With the Replicated SSAC, *get_for_read* that hits in its local representative is handled identically to that of the Shared SSAC. However, on any miss or *get_for_write* and *put* pair, additional work must be done to maintain the directory and the consistency of any copies that exist.

The Clustered Object classes makes it easy to replicate the Simple Shared Array SSAC into multiple representatives using the *MHReplicate* Miss Handling class. Most effort is needed to implement the coherence actions between the representatives. In particular, it is necessary to add the global directory and to extend the *get* and *put* operations to make use of the directory.

In one implementation of the counter (section 4.1.2), a shared array was used to maintain pointers to the representatives. The array was relatively small and instantiated on the processor on which the first representative was created. Additionally, accesses to the array did not need any explicit synchronization, since after being filled, the array was accessed read only. As such, accesses to it were efficient, as each processor would locally cache the array in the processor cache. This is not as simple in the case of the directory, for the directory can be of considerable size considering that it needs to track every cache entry. Access to the directory will require explicit synchronization to ensure its correctness, and it may often require updates. Allocating the directory on one processor will imbalance the per processor memory usage of the Simple Replicated Cache and will also lead to higher contention on the memory module in which it resides.

By partitioning the directory across the representatives, its memory usage can be equally distributed. Each representative would thus contain a portion of the directory. By also dividing the

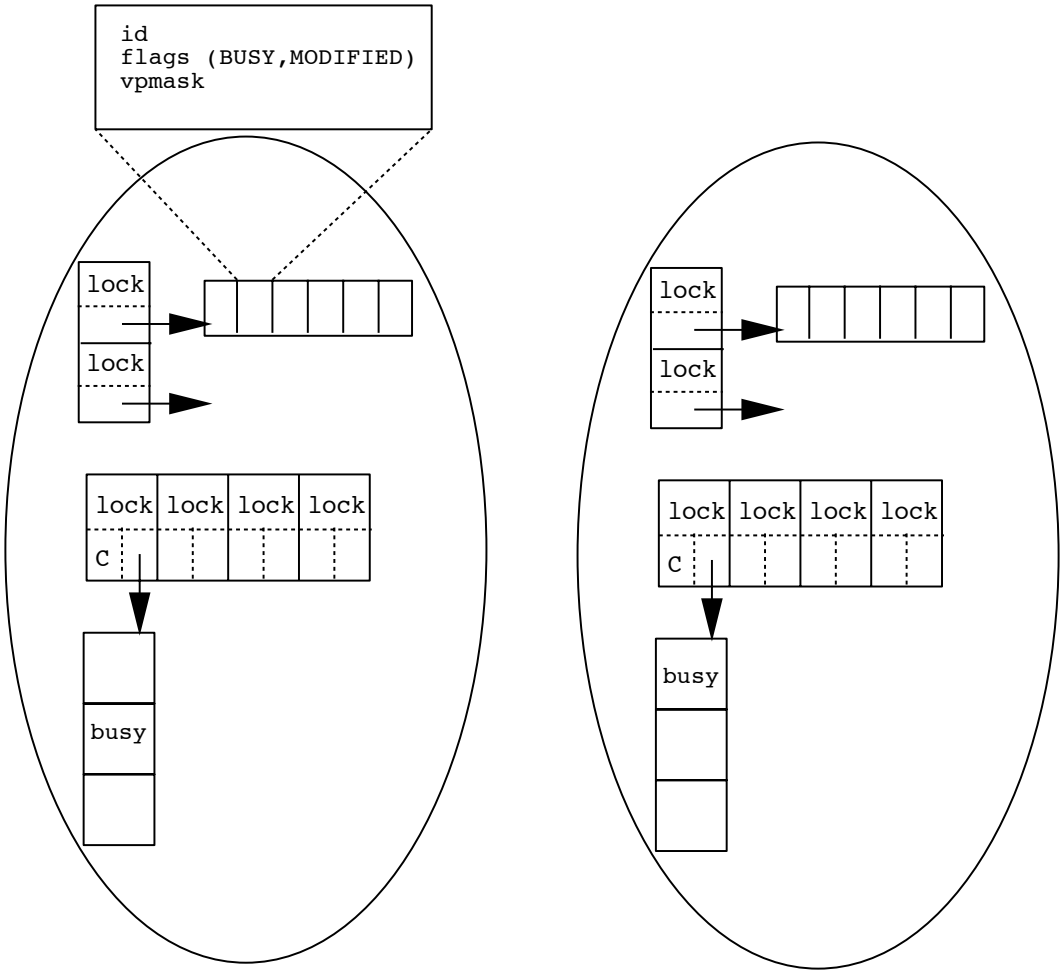


Figure 4.8: Replicated SSAC composed of two representatives.

number of hash queues by the number of representatives, it is possible to assign each hash queue to a particular directory partition. The directory partitions can either be directly embedded into each representative, or the entire directory can be implemented as a separate clustered object. Access to the directory can be implemented either through shared memory or remote procedure invocation. The example implementation of figure 4.8 directly embeds the directory partitions into the representatives of the Replicated SSAC and uses shared memory to access and update the portions.

The example makes use of the convenience method *FindRepOn* of the *MHReplicate* misshandler when the representative associated with a specific processor must be located. Alternatively, it could have been written to use a global array of representative pointers as in the *CounterArrayCO* example of the previous section. However, it was felt that additional optimizations should not be used so as to allow a comparison favoring the Shared implementation. Figure 4.8 illustrates a Replicated SSAC that is composed of two representatives. Each representative maintains its own

hash table as well as a portion of the directory.

It is clear that the performance of the Replicated SSAC will depend on the amount of coherency traffic required by the given access pattern. The more copies that exist of an element and the larger the number of write requests, the more time will be spent doing coherency maintenance. Moreover, maintaining coherence requires remote memory accesses, further adding to the costs. On the other hand, if the access pattern is dominated by *get_for_read* requests or if the processors access independent COIDs, then the costs associated with replication will pay off due to increased data and cache locality.

4.2.3 Partitioned SSAC

Another way to organize the SSAC into multiple representatives is to partition the hash table, where each representative is given responsibility for some exclusive portion of the COID space and corresponding portion of the hash table. All COIDs are said to be owned by a specific representative and hence the processor it is associated with. The owning representative is responsible for caching and maintaining all state associated with its COIDs. Figure 4.9 illustrates such an organization.

The hash function now maps a COID to a Processor Index and a Queue Index. The Processor Index is used to identify the target representative. The Queue Index determines the target hash queue within the owning representative. With this organization, only one copy of a cache object exists at any one time, so no additional directory or coherency is required.

Local *get* and *put* operations function identically to the Shared SSAC operations. Remote operations can use either shared memory or remote procedure invocations. The example implementation uses shared memory.

Similar to the shared memory counter examples of the previous section, a shared memory implementation requires that one representative be capable of accessing the data of another representative. The use of a shared array of representative pointers or the Miss Handling classes convenience function *FindRepOn* seem most appropriate for locating the remote representative. The example implementation uses the *FindRepOn* operation. The *get* and *put* operations can then be seen as simple extensions to the Simple Shared Array SSAC operations. Both operations simple apply the hash function to obtain both the Processor Index and the Queue Index. The Processor Index is used with the *FindRepOn* method to locate the appropriate representative. The operations then proceeds as before with respect to the identified representative's data. Since it is possible that a representative may not exist for a given processor, it is necessary to check the validity of the

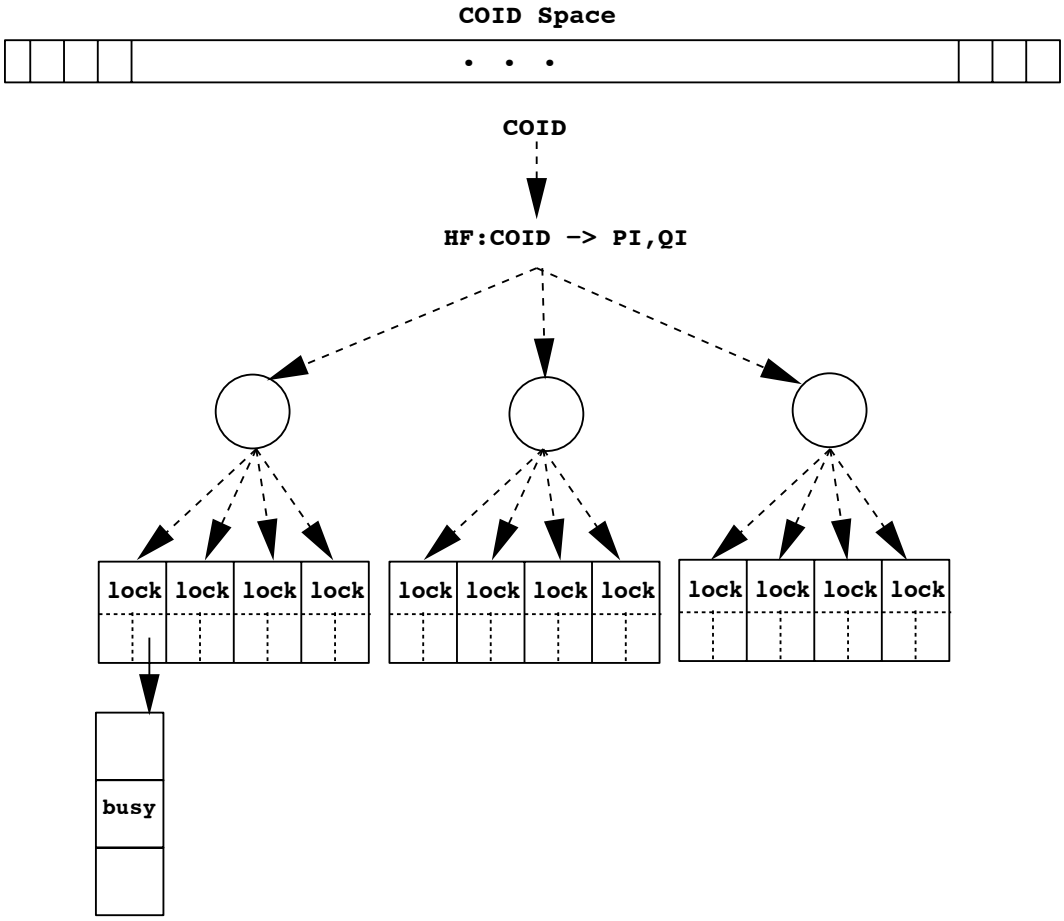


Figure 4.9: Partitioned SSAC

pointer in the shared array. If it is not valid, a request on the remote processor must be made to create a representative.

With a remote procedure call approach, remote procedure calls are used to direct the call appropriately after having determined that the request should be made to a remote processor. From an implementation point of view, the remote procedure call approach is simpler than the shared memory approach.

The performance of the Simple Partitioned Array SSAC will depend on the locality of the requests, as will be seen in Chapter 5.

4.2.4 Summary

The SSAC architecture was designed to have good lock performance. However, it was not optimized for data or cache locality. In the above subsections, three different Clustered Object implementations of the SSAC architecture were presented. The first Clustered Object implementation is a straight-forward shared version. It illustrates that a Clustered Object implementation need not impose any additional complication per se. The next two implementations show that locality optimizations can be expressed within the Clustered Object framework. Using Clustered Objects, one can extend the standard implementation to use multiple representatives in order to increase data and cache locality. Chapter 5 will compare their performance characteristics.

Chapter 5

Performance

Clustered Objects are designed so that locality management optimizations can be implemented in a standard and reusable manner. There are two main performance goals for Clustered Objects in general. First, a Clustered Object implementation should be able to achieve the same performance as the corresponding non-Clustered Object implementation. Secondly, if a useful library of Clustered Objects is to be developed, then Clustered Objects must be able to provide different performance tradeoffs for different access pattern while maintaining a common interface.

A definitive evaluation of whether Clustered Objects can achieve these two performance goals is beyond the scope of this dissertation. However, as an initial step, two series of experiments were carried out on the Clustered Object implementations presented earlier. The first set of experiments compares the performance of the non-Clustered Object implementations of the counter to the corresponding Clustered Object implementations for one specific access pattern. The second set of experiments compares the three Clustered Object implementations of the SSAC architecture using three different access patterns. Our results indicate that there is promise in Clustered Objects being able to meet the stated performance goals.

5.1 General Experimental Setup

The Tornado kernel provides the software base for all of our experiments. The Tornado software base runs on both the NUMAchine hardware platform and the Stanford SimOS software platform [41, 31, 30]. The Clustered Object class library was implemented on top of Tornado's Object Translation System. Each experiment was implemented as a custom kernel that set up and performed a specific test. Kernels were used, rather than user level applications, to reduce costs when run as simulations.

Each kernel was run on a range of one to sixteen processors.

5.1.1 SimOS and NUMAchine

The Stanford SimOS simulator, developed by Rosenblum et al. [31, 30], is a complete, machine level, multi-processor simulator. It models the processors, caches, memory system and I/O devices of a multiprocessor system. The Tornado project uses SimOS as a platform for both development and performance analysis. SimOS is able to simulate the hardware components with sufficient detail such that it can be used to execute the same binary versions of the Tornado kernel that execute on the NUMAchine hardware. SimOS makes it possible to attribute execution costs to hardware events without the addition of instrumentation into the code being tested.

At the time of this work, the NUMAchine platform was in its last stages of development. There was opportunity to run a limited number of the experiments on a sixteen processor system. These experiments were used to provide a degree of validate to some degree the results obtained with the SimOS simulator.

5.1.2 Simulated Machine

The experiments use a configuration of the SimOS simulator that models a general cache-coherent NUMA multiprocessor. While the simulated machine architecture is based on the NUMAchine platform it does not implement any of NUMAchine's specific optimizations, such as its network caches or its novel coherency protocol [41].

The simulated machine is composed of 16 64-bit R4400 MIPS processors[26], organized into 4 processor stations, connected by a ring network. Each processor has separate 16KB primary data and instruction caches[1] and a 1MB 2-way set associative unified secondary cache. The primary cache line size is 32 bytes and the secondary cache line size is 128 bytes. Each station is configured with 64MB of memory. The memory on each station is transparently accessible by all processors in the system. Accesses to off-station memory, however, exhibit longer latencies. The minimum uncached off-station memory access is approximately 3.5 times longer than an on-station memory access[2].

[1] Both Primary caches were configured as one way set associative for the Counter tests and as 2 way set associative for the SSAC tests.

[2] This value does not take into account any delays due to contention.

5.2 Experiments

The goal of our experiments is to gain insight into the performance of the Clustered Objects described in Chapter 4 under synthetic workloads. Synthetic workloads make it easier to identify and control the parameters of the workload and understand their effect on performance. The experiments thus do not attempt to realistically model or characterize "real" workloads.

Each experiment is composed of a series of runs, where each run:

1. instantiates the Clustered Object under examination;
2. makes an initial request to the Clustered Object on each processor, ensuring initialization of the translation entries;
3. starts a worker process on each processor; and
4. runs all workers concurrently, making a fixed number of requests to the Clustered Object on each processor.

We measure the number of cycles spent executing the workers in step 4. The sum of the cycles is considered to be the total cost for the fixed number of requests to the Clustered Object. Dividing the total cost by the total number of requests yields a per request cost, expressed in processor cycles. Initialization and termination cost are not included. To ensure that no worker unfairly benefits from warm hardware caches, all hardware caches are cleared prior to starting the workers. As a result, the costs for the initial cold cache misses will be included in the results.

Ideally the number of cycles per request would be the same regardless of the number of processors concurrently accessing the Clustered Object. However, synchronization, sharing and algorithmic costs can all grow as the number of processors is increased, causing increases in the cost per request. An implementation is considered more scalable if the costs associated with an increase in processors are smaller. However, it should be noted that scalability of an implementation will vary with the access pattern of the workload.

Using the features of SimOS, the number of cycles required per request are broken down as follows:

- Cycles spent stalling on accessing data:

  dStallRemote: Cycles spent waiting for off-station data accesses to complete.

  dStallLocal: Cycles spent waiting for local data accesses to complete.

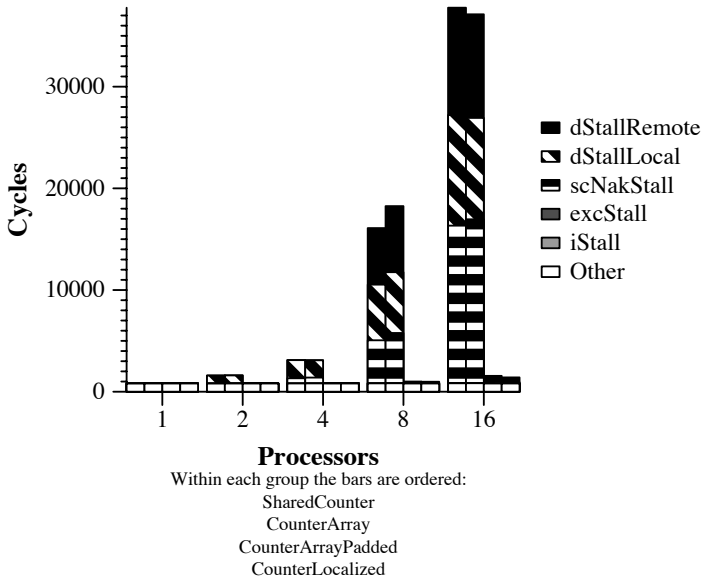


Figure 5.1: Non-Clustered Object Counter: performance results obtained with SimOS.

scNackStall: Cycles spent waiting for Store Conditional to be acknowledged.

excStall: Cycles spent waiting to gain exclusive access to a cache line.

- Cycles spent stalling on Instruction accesses:

  iStall: Cycles spent waiting for instruction accesses to complete.

- Cycles spent executing Instructions:

  Execute: This represents the cost of executing the actual instructions that compose the Worker processes. Cycles spent spinning on locks are included in this amount.

5.2.1 Counters

This subsection first revisits the results presented in the Background and Motivation chapter. By running the tests of the non-Clustered Object counter implementations on both SimOS and the NUMAchine hardware a degree of validation for the results SimOS is obtained. Then using SimOS the performance of the Clustered Object implementations of the counter are compared to the performance of the non-Clustered Object implementations.

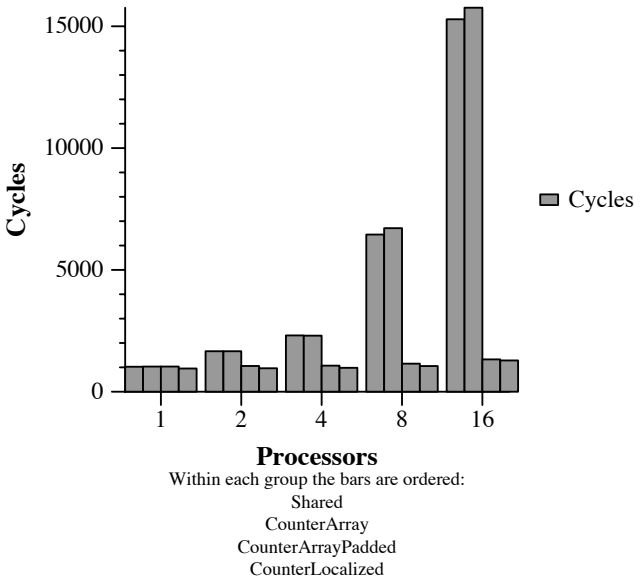


Figure 5.2: Non-Clustered Object Counter: performance results obtained with NUMAchine.

SimOS vs NUMAchine results

The Background and Motivation chapter presented a Counter data structure that was used to illustrate locality optimizations. Four non-Clustered Object implementations were presented along with their performance under a specific access pattern. The *SharedCounter* implementation used a single shared atomically updated integer value. The *CounterArray* used an array of atomically updated integer values with one value per-processor. The *CounterArrayPadded* was identical to the *CounterArray* but padded each integer value to a secondary cache line boundary, eliminating false sharing. Finally *CounterLocalized* also used per-processor integer values but ensured that each processor's value was located in its local memory. Each implementation was tested with a constant number of requests; 1% of all requests were to obtain the total value of the counter and all other requests were either increments or decrements. The results obtained with SimOS are summarized in figure 5.1. These same tests were also run on the NUMAchine hardware, the results of which are presented in figure 5.2. On NUMAchine the timing was obtained by adding instrumentation prior to the start and end of each worker. From the statistics obtained, a cycles per request cost was calculated.

The figures show that although the absolute values of the SimOS and NUMAchine runs (figures 5.1 and 5.2) differ by a factor of two, the general trends are similar; the *CounterPaddedArray* and *CounterLocalized* implementations outperform the other implementations in both figures in similar

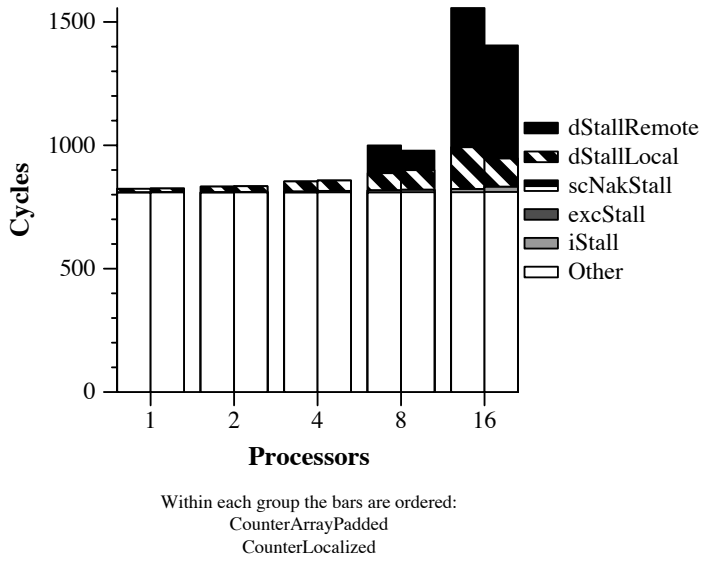


Figure 5.3: Comparision of *CounterArrayPadded and CounterLocalized.*

proportions. This confirms that a relative performance difference illustrated by the simulator is likely to be experienced on a real hardware platform. The remainder of this chapter will focus on the results obtained from SimOS.

Figure 5.3 shows the difference between the *CounterArrayPadded* and *CounterLocalized* in more detail. The *CounterLocalized* implementation performs slightly better, primarily due to slightly lower remote data stall time. This behaviour is to be expected: placing the local counters in the memory of the stations closest to the processors accessing it, reduces the number of initial remote misses. The remaining remote misses are caused by the 1% of requests for the global counter value. The performance results of the non-Clustered Object *CounterLocalized* will be used as a reference point for comparing to the Clustered Object versions.

Clustered Object Alternatives

The Examples chapter presented a number of Clustered Object implementation of the Counter data structure. Figure 5.4 presents the results for each implementation. The left-most bar in each group is the performance of the non-Clustered Object *CounterLocalized* implementation, the most efficient of the non-Clustered Object implementations. The performance for the *CounterRemoteCO* is not shown in the graph, as Tornado currently does not support the large number of remote procedure calls required. Moreover, as stated in chapter 4, we do not expect *CounterRemoteCO* to perform well in any case, as the cost of each remote procedure call will make the global value method

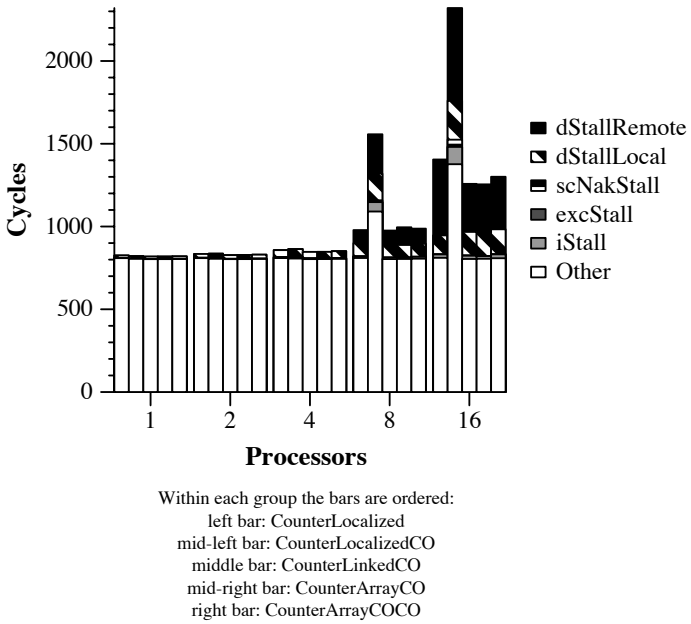


Figure 5.4: Non-Clustered Object Counter Performance results obtained with NUMAchine.

extremely expensive.

All the Clustered Object versions, with the exception of *CounterLocalizedCO*, perform as well as the non-Clustered Object implementation. The *CounterLocalizedCO* is implemented using the *MHReplicated* Miss Handling convenience functions. As stated earlier, these functions are not optimized for representative use at this time and hence suffer a considerable performance penalty when used by the representatives.

These results do not indicate, and are not meant to show, that Clustered Object implementations inherently outperform non-Clustered Object implementations. To the contrary, a developer is certainly capable of directly implementing any optimization that Clustered Object might make. The key advantage of using Clustered Objects, however, is that it provides the programmer with a standard way of implementing the optimizations in a reusable manner. From the results, it is evident that Clustered Objects can achieve performance as good as any non-Clustered Object implementation.

5.2.2 SSACs

In Chapter 4 we presented three Clustered Object implementations of the SSAC data structure. The goal of the experiments in this section is to show that the different Clustered Object implementations can provide different performance tradeoffs. This work does not attempt to provide a

complete characterization of realistic SSAC workloads. Similarly, it does not attempt to provide a detailed performance analysis of each of the three implementations. Instead, the experiments intend to show that a given accepted multiprocessor data structure, which has already been tuned to provide low contention, can still benefit from the locality optimizations that are supported by Clustered Objects. No one implementation is able to perform well under all access patterns. Thus, having multiple implementations supporting an identical interface, however, allows a developer to choose the right implementations for the access pattern expected.

In all of the SSAC experiments, a fixed number of requests are generated as part of the initialization of the experiment. Each request is either a read request or a write request for a specific Cache Object. The proportion of read to write requests and the generation of COIDs to identify the requested Cache Objects are specified as parameters of the experiments:

Fraction write: fraction of total requests that are write requests.

Fraction local: fraction of requests for COIDs that map to those assigned to the local processor. (described in more detail below).

The *fraction local* parameter controls the degree of locality in the request pattern. By assigning each processor a specific sub-range of COIDs, it becomes possible to control the number of requests to the sub-range specific to the requesting processor. If *fraction local* is set to 0 then all requests are randomly chosen from the entire range of COIDs and hence there is no explicit locality in the requests. However, if *fraction local* is set to 1 then all requests on a given processor will be to COIDs that are in the sub-range assigned to the processor on which the request was made. For example, assume the total range of COIDs is 0-127 and the number of processors in the experiment is 4 and the range is divided into 4 equal non-overlapping sub-ranges; 0-31, 32-63, 64-95, and 96-127. If *fraction local* is set to 0 then all request on all processors are chosen randomly over the entire range of 0-127. On the other hand if *fraction local* is set to 1 then all the requests on each of the processors is chosen from the sub-range assigned to the processor, eg. all requests on processor 0 would be in the range of 0-31, all requests on processor 1 would be in the range of 32-63 and so on.

We consider three scenarios with the *fraction write* and *fraction local* set as follows:

Case 1 Fraction write is 0 and fraction local is 0.

Case 2 Fraction write is 0.05 and fraction local is 1.

Case 3 Fraction write is 0.05 and fraction local is 0.

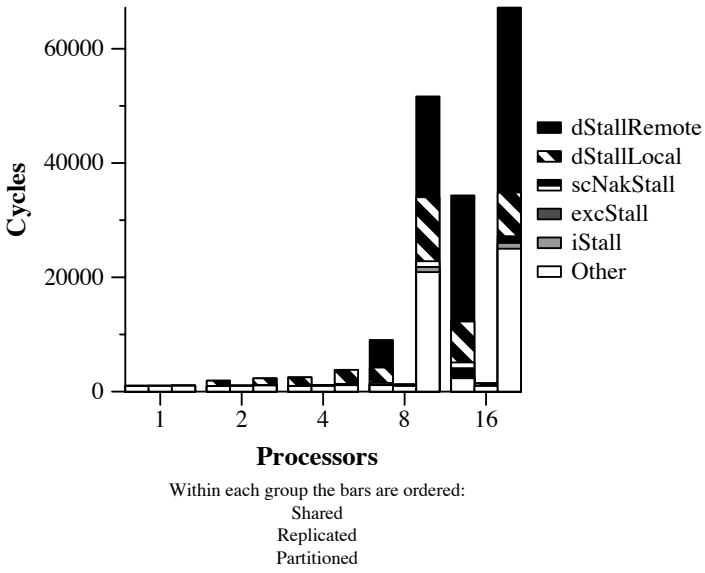


Figure 5.5: Case 1: Random read requests.

Additionally, in order to ensure no cold misses occur, the SSACs are pre-initialized with the appropriate Cache Objects whose COIDs span the range of COIDs. In cases 1 and 3, the SSACs are initialized so that the entire range of COIDs have been requested at least once by each processor. In case 2, the SSAC is initialized so that each processor has requested its local range of COIDs once. In all cases the SSAC is large enough to hold all elements. The results of these experiments thus represent what one would expect of the SSAC implementations once a steady state has been reached.[3]

Figures 5.5, 5.6 and 5.7 show the results for the three separate access patterns, respectively. Each graph shows how the three SSAC implementations compare with each other for one of the specific cases.

In figure 5.5, the Replicated SSAC clearly outperforms the other two implementations. Since there are no write requests and no misses, the Replicated SSAC need not perform any coherency. In this case, the cost of replication is well worth it. Each Cache Object that was replicated during initialization can be reused. The random nature of the requests does not have any negative effects for the Replicated SSAC. However, the Shared SSAC suffers from an increase in data misses due to the real and false sharing induced by the random accesses. The Partitioned SSAC not only suffers from an increase in data misses due to the random access, but also from the additional costs of

[3]Removing misses from the access patterns was done for simplicity, as performance tradeoffs can be observed without the introduction of misses.

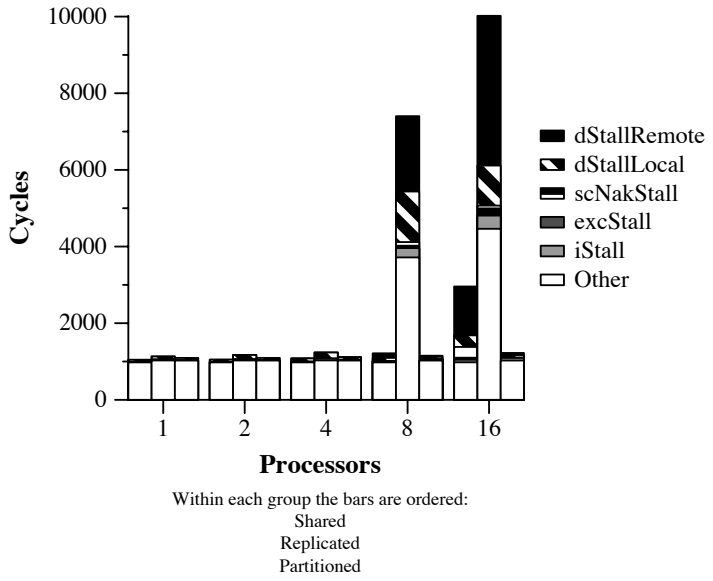


Figure 5.6: Case 2: Local requests with 5% being writes.

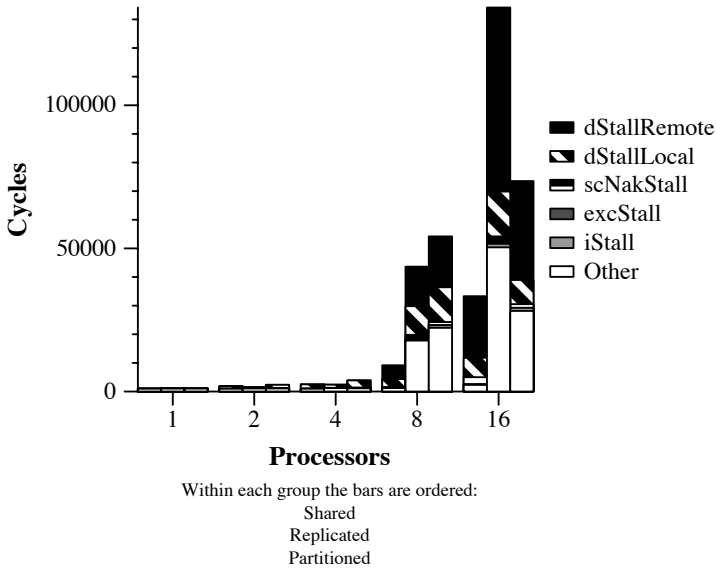


Figure 5.7: Case 3: Random requests with 5% being writes.

identifying and locating remote COIDs. This is reflected in the increase of cycles attributed to 'Execute'.

For Case 2, where all requests are local and five percent are writes, the Shared and Partitioned SSACs are able to exploit the high degree of locality in the access pattern without suffering additional costs for the write requests (figure 5.6). The performance of the Shared and Partitioned SSACs are similar up to eight processors. At sixteen processors, the Shared SSAC displays an increase in cycles spent stalling on data accesses due to false sharing. The sizes and alignment of the data elements of the Shared SSAC do not cause false sharing with fewer than 16 processors; the false sharing is avoided by the Partitioned SSAC, as the individual portions of the underlying array of hash queues are physically separated in the address space.

The Replicated SSAC performs poorly with respect to the other two implementations. For each write request, considerable coherency overhead is necessary while replication provides little benefit. The Replicated SSAC must first locate the representative that contains the directory entry for the COID. Since the *MHReplicated::FindRepOn* method is used to locate representatives, off-station accesses to the processor on which the Miss Handler exists will typically be required. Furthermore, since the directory is evenly distributed across all the representatives, there is no guarantee that the entry for the target COID will be local to the processor, leading to additional remote accesses. Finally, since each processor accesses its own Cached Objects, the directory will always indicate that there is only one copy of the Cache Object, namely on the processor (on which the write request was issued).

Clearly, optimizations could have been used to reduce these costs. For example, the use of an exclusive flag along with a write-invalidate protocol would have suffered lower coherency overheads, as each write request would then occur exclusively. Another optimization would be to eliminate the use of *MHReplicate::FindRepON*, as was done in some of the Counter Clustered Object implementations. It is important to note that although optimizations may reduce some of the overheads, they cannot eliminate the fact that the Replicated SSAC requires extra overheads to maintain coherency, while the Shared and Partitioned SSACs do not. Additionally, a given optimization may only be relevant for some access patterns and as such may be better served by a separate Clustered Object implementation. For example, the fact that the write-invalidate protocol does not perform well under case 2 does not mean that it is categorically inappropriate. Stenström showed that under some access patterns, write-update protocols have better performance, whereas write-invalidate protocols are superior under others [34]. This argues that rather than changing

the current implementation of the Replicated SSAC to use write-invalidate, it would be better to add a new Clustered Object that specifically targets the access pattern in question.

Finally for case 3, where the requests have little locality and five percent are writes, the Shared SSAC provides performance better than the other two implementations, as shown in figure 5.7. In this case, the write requests combined with the completely random choice of COIDs forces the Replicated SSAC to suffer the greatest possible coherency overheads. Each representative will contain a copy of each of the Cache Objects being accessed and as such will require explicit updating on every write. The costs for updating the copies are not amortized over any significant number of requests and as such result in no benefit. The random nature of the requests causes the Partitioned SSAC to perform similarly as in Case 1.

Summary

We have shown that each of the three SSAC implementations is able to address a specific access pattern more effectively than the other two: the Replicated SSAC performs best for case 1; the Partitioned SSAC for case 2; and the Shared SSAC for case 3. While the access patterns chosen are not necessarily representative of realistic workloads and do not cover the entire space, they do represent three distinct workloads that place differing demands on the SSAC implementations. The three Clustered Object implementations each maintain the same interface and yet display separate performance characteristics. Booch points out that a foundation class library should be "Complete" in that:

> The Library must provide a family of classes, united by a shared interface but each employing a different representation, so that developers can select the ones with the time and space semantics most appropriate to their given application.[4]

In this light, the results provide initial evidence that Clustered Objects may be suitable for the development of a "Complete" multiprocessor foundation class library. It should be possible to develop implementations that provide various tradeoffs in locality management for differing access patterns.

Chapter 6

Design Guidelines

We found that when implementing Clustered Objects, it is generally best to adhere to the following guidelines described in more detail in the following subsections:

- One should ensure that the most common operations are optimized for locality.
- A family of Clustered Objects should be implemented when different internal policies better support different access patterns.
- Representatives that frequently access each other should directly maintain references to the representatives they access.
- One should consider implementing separate Clustered Objects for logically separate entities that can have different data management policies.
- Internal Padding should be used to ensure a secondary cache line boundary between representatives.

6.1 Optimize Most Common Operations for Locality

Generally, a specific implementation of a given data structure will not perform well under all access patterns. Hence, the expected use of the data structure should be taken into account when determining what operations should be optimized for locality. For example, in the case of the integer counter, it was stated that the most frequent operations are modifications of the counters value. For this reason, all distributed implementations ensured that the *increment* and *decrement* methods are performed entirely with local memory accesses, even though the *value* method then

became primarily a more expensive remote memory access operation. This is not a problem if the *value* method is invoked infrequently relative to *increment* and *decrement*. If, however, the global *value* method were invoked more frequently, then perhaps *value* should have been optimized for *locality* (for example by treating each representative's local value as a replica of the global value and using some form of coherence protocol when modifying the counter's value). Of course, it is possible that the workload might invoke the modification and *value* operations equally, in that case a shared or partially shared implementation may prove more appropriate.

6.2 Provide Multiple Implementations

When developing a Clustered Object, it becomes necessary to choose one policy over another when implementing a particular internal function. For example, in the case of the Replicated SSAC one must choose between write-update, write-invalidate or some other protocol when implementing the internal coherence of the replicated values. The protocols yield different performance results for different access patterns. In such cases, it is best to provide multiple versions of the Clustered Object so that a programmer can choose the right implementation for the access pattern expected.

6.3 Split Complex Clustered Objects into Multiple Clustered Objects

The different components of a complex Clustered Object could each potentially use a separate data management policy. For example, the replicated SSAC consists of two key components: the hash tables and the directory. It is not obvious that the same data management policy is appropriate for both components, under any given access pattern. Although the hash tables are replicated in our example, the directory need not be, and could instead be shared or partitioned. The use of a separate Clustered Object to implement the directory allows for greater flexibility and customization. One might implement a simple shared directory or some partitioned directory Clustered Objects, all with the same external interface.

Decomposition of software into distinct interacting objects, with fixed interfaces, is fundamental to the development of a runtime customizable system like the ones proposed by Krieger et al. [2] and Bershad et al. [3]. Using separate Clustered Objects for each component naturally leads to composition at run time. For example, the Replicated SSAC could be implemented to take a

Clustered Object Identifier of its directory Clustered Object as an instantiation parameter, allowing the instantiator of the Replicated SSAC to specify what type of directory to use by first instantiating the appropriate directory Clustered Object and then passing its Clustered Object Identifier to the Replicated SSAC.

6.4 Maintain Inter-representative References

If representatives need to locate each other frequently, they should directly maintain the necessary representative references. For example, the representatives of the *CounterLinkedCO* and *CounterArrayCO* Clustered Objects both directly maintained pointers to the other representatives. Both implementations did not need to use the Misshandling representative management functions and thus performed better than the *CounterLocalizedCO* implementation.

In cases where the operations of a representative need only access a limited number of neighboring representatives, it is best to explicitly organize them with embedded pointers. This leads to standard organizations of representatives in linked lists, queues, trees, etc. On the other hand, if operations of the Clustered Object need to randomly access representatives or iterate over all representatives, a shared structure such as an array of pointers would be more appropriate. A shared array structure is simple to update and has better spatial locality. However, a shared array can lead to increased cache misses if the array is frequently updated.

6.5 Pad Representatives

It was generally found that for high performance, it is necessary to pad the representatives to secondary cache line boundaries. This ensures that false sharing between the data elements of one representative and other data does not occur. However, it can lead to considerable memory overhead per Clustered Object. In the worst case, a Clustered Object is implemented with a clustering degree of 1, and each representative contains one word of data, in that case 120 bytes of overhead would result per-representative, assuming 64-bit words and 128-byte secondary cache lines.

Chapter 7

A new Clustered Object Model

This work represents just an initial attempt at using the Clustered Object approach and certainly does not constitute any final design. In implementing the Clustered Objects, we learned a number of lessons and today we would probably implement Clustered Objects differently. This chapter presents a new Clustered Object model motivated by the experience gained.

7.1 Limitation of initial Model

At the outset, we thought we would design and then provide a class foundation library that would ultimately serve as the means by which a programmer would implement new Clustered Objects and that it would be based on Tornado's object reference translation and miss handling support. The library would provide pre-built patterns of commonly occurring Clustered Object designs.

We had hoped that by building some initial Clustered Objects with the existing infrastructure, the necessary insight would be gained to guide the development of the library. During the course of our work, it became apparent that a well defined model for Clustered Objects was necessary, along with a class representation to support it. Basing the model on Tornado's Object Translation System led to a simple model in which a Clustered Object is composed of a Misshandling Object and a set of representative objects. This model was easily represented with two class hierarchies. However, this initial model, although simple, has a number of shortcomings:

- It does not clearly separate Clustered Object management from Misshandling behaviour. The Misshandling object is used both as a generic interface to the Object Translation System and to manage and connect the components of the Clustered Object.

- It does not support Clustered Object global data in a standard way. The data members specified in the representative class are local to each representative instance. There is no natural way to indicate that some data members should be global to all representatives of a particular Clustered Object, but not allow access from other Clustered Objects.

- It does not support initialization parameters for Clustered Objects in a standard way. When a Clustered Object is instantiated only the Misshandling object is created, the representatives are instantiated on first use by the Misshandling object. Thus a means for communicating initialization parameters to the representatives, specified when the Clustered Object is instantiated, is required.

- It lacks a clear separation of a Clustered Object's distinct interfaces. A Clustered Object has two types of interfaces:

  - an external client interface.
  - internal interfaces between the constituent objects. Eg. inter-representative interfaces.

  Without support for multiple separate interfaces it is difficult to specify the independent roles of an object. One cannot easily specify a new Clustered Object that both supports a predefined external interface A and a predefined inter-representative interface B.

7.2 New Model

Figure 7.1 illustrates a new model that attempts to address these shortcomings. It was developed based on the experiences we have gained so far, but it has not yet been implemented. In Figure 7.1, four different types of arrows are used:

Instantiates: the object at the tail of the arrow instantiates the object at the head of the arrow.

Invokes Methods of: the object at the tail of the arrow invokes specific methods of the object at the head of the arrow. An object at the head must implement a specific interface for each 'Invokes Methods of' arrow that points to it.

Contains a pointer to: the object at the tail of the arrow contains a pointer to the object at the head of the arrow.

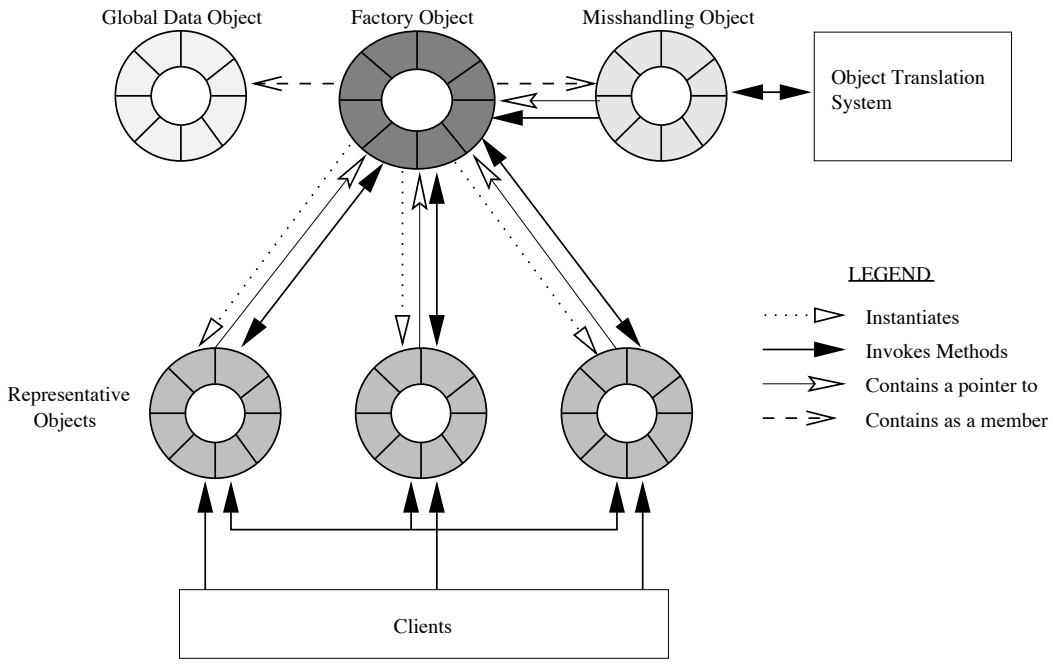


Figure 7.1: A New Model of Clustered Objects: The Factory Model

Contains as a member: the object at the tail of the arrow contains the object at the head the arrow, as a data member. Objects contained within another object are created when the containing object is instantiated.

7.2.1 Factory Object

In the new model, every Clustered Object contains a Factory object. The Factory object acts as the centralized manager of the Clustered Object. Its primary responsibilities are to create all constituent objects, initialize them as necessary, and provide centralized methods and data. In the new model, a Clustered Object is created by instantiating the appropriate Factory object. As can be seen by the 'Contains as a member' arrows in figure 7.1, the Factory object contains both a Misshandling object and a Global data object. Both of these will be discussed in the following subsections. Since they are contained within the Factory object as members, they will be created along with the Factory object. As illustrated, the Factory object also instantiates all representative objects.

The Factory Object implements a specific interface that can be invoked by the Misshandling object when representatives need to be created or destroyed. The Factory object also implements interfaces invoked by the representative objects. These methods can be invoked by any represen-

tative of the Clustered Object and are largely left to the implementor to define as required. An example of two standard methods that might compose this interface, are *lock_rep_creation* and *unlock_rep_creation*, which suspend the changes to the set of representatives when accessing the entire set.

It is expected that the class representation for the new model will provide a separate hierarchy of Factory objects. Each class will provide a base definition for a Factory object. When writing a new Clustered Object, an implementor will create a new Factory Object by inheriting from one of the classes in the Factory hierarchy.

7.2.2 Misshandling Object

The Misshandling object in the new model acts solely as an interface between a Clustered Object and the Object Translation System. It implements the necessary methods required by the Object Translation System, namely *handlemiss* and *cleanup*. It keeps track of which representatives have been assigned to which processors and enforces the clustering degree of the Clustered Object.

The Misshandling object has a pointer to the Factory object, as illustrated in figure 7.1. The pointer is required so that the Misshandling object can make requests to the Factory object in order to create new representatives. The Misshandling object is unaware of what class the representatives belong to or how they should be initialized. It expects the Factory object to implement any Clustered Object specific knowledge, such as which class to instantiate representatives from, how to initialize them and potentially connect them.

It is expected that one Misshandling class will be sufficient to efficiently support all Clustered Objects and it would be provided as part of the basic infrastructure. All Factory object classes would contain a member of this Misshandling class by definition.

7.2.3 Global Data Object

In the initial model there was no standard notion of data that is accessible by all representatives within a Clustered Object. The data members of a representative are local to the representative, and modifications to one representative's members do not affect the members in other representatives. Of course, explicit coherency could be implemented between the representatives to ensure consistency between a subset of their data, but this adds to complexity and overhead and is only warranted in some read-after-modify circumstances. It seems more appropriate to use shared memory to implement simple data that should be globally accessible by all representatives.

```
class foo : public Factory
{
    GLOBAL:
       int _repcount
    private:
        .
        .
        .
};

void
foorepclass :: bar ()
{
   int a=_repcount;
     .
     .
     .
}
```

Figure 7.2:

In the new model, each Clustered Object has a Global Data object embedded within the Factory object. The Global Data object contains the global data members of the Clustered Object. As illustrated, each representative has a pointer to the factory object and thus can gain access to the Global Data object. Clearly, any of the Factory objects members could be treated as global data. However, providing an explicit object for all global data allows the global data members to be identified separately from the other data members of the Factory object. For example, C++ access rules dictate that data members of a class are not accessible from outside the class, unless another class is granted special access through the friend construct. To provide the representatives of a Clustered Object access to some of the data members of the Factory Object, the class of the representatives would have to be explicitly granted friend access, but this would give the representatives access to all the data members of the Factory object and not just the ones intended.

Ideally, there would be compiler support for a special access type called *GLOBAL* that could be used to identify data members as being accessible to the representatives of a Clustered Object. Figure 7.2 gives an example of how one might define a Clustered Object *foo*, which has a Global data member *_repcount*, assuming a compiler supported the *GLOBAL* access specifier. If the representative class for the Clustered Object *foo* is *foorepclass*, then the figure also illustrates a method of the representative class called *bar* that makes use of the global data member *_repcount*.

Since compiler support for *GLOBAL* does not exist, a separate global data object, along with a set of macros, can be used to provide similar functionality. Figure 7.3 illustrates the same example

```
class foo : public Factory
{
   GLOBAL_START(foorepclass)
      int _repcount;
   GLOBAL_END
       .
       .
       .
};

void
foorepclass :: bar()
{
   int a=GLOBAL(_repcount);
     .
     .
     .
}
```

Figure 7.3:

```
class foo : public Factory
{
   public:
      class Globals {
         friend class foorepclass;
         int _repcount;
      } globals;
   private:
       .
       .
       .
}

void
foorepclass :: bar()
{
   int a=_factory->globals._repcount;
     .
     .
     .
}
```

Figure 7.4:

using a set of macros. Figure 7.4 shows how the macros could be expanded to implement the global data via a separate Global data class. Data members declared between the macros, called *GLOBAL_START* and *GLOBAL_END* are considered to be global members. The macros expand to define a *Globals* class within the scope of the Factory object. This initial class explicitly declares the representative class, *foorepclass*, as a friend, thus granting the representatives access to the Global Data object's members. The macros also ensure that the Clustered Object contains an instance of the *Globals* class called *globals*. Finally, the implementation of the *bar* representative function now uses a macro called *GLOBAL* to access the *_repcount* global member. The *GLOBAL* macro expands to ensure that the access to the global data member happens via the representative's pointer to its Clustered Object's Factory object.

7.2.4 Initialization Parameters

In our current class representation, there is no standard support for client specified initialization. When client code instantiates a Clustered Object, it often needs to specify a set of initial parameters that need to be recorded and passed (or made accessible) to the representatives. For example, a Clustered Object that implements some sort of array, may require the user to specify the size of the array at creation time. This value needs to be available to the representatives so that they can allocate their local resources appropriately. Similarly, the same Clustered Object may allow the instantiator to specify a parameter that indicates whether the Clustered Object is to implement a partitioned or replicated array. Again, the representatives need to be aware of the value of the initialization parameter to behave appropriately.

In the new model, initialization parameters are treated as global data members. If the representatives require their own copy of the initialization parameters, they are free to make a local copy in their constructor. This is also true for any of the other global data members.

7.2.5 Representative Objects

In both the initial and the new model, the representatives serve as local implementations of the Clustered Object's external interface. In the initial model there was no explicit separation of internal and external interfaces.

The new model explicitly identifies the different interfaces implemented by the components of the Clustered Object. This is particularly relevant to the representatives as they implement the majority of the interfaces. Figure 7.1 shows three 'Invokes methods of' arrows pointing to each

representative. The most obvious interface is the external client interface. These are the methods that the clients see as the uniform interface to the Clustered Object. Another interface is the internal inter-representative interface. These are the methods that representatives may invoke of each other. The final interface is a representative management interface for use by the Factory object. For example, the Factory Object invokes a method on each representative to set the representative's Factory Object pointer.

The main advantage of separating the interfaces is that it naturally leads to separate hierarchies for each interface. In the new model, one expects three different hierarchies of interfaces. One hierarchy would define all the external interfaces a Clustered Object might support. This hierarchy will grow with every new Clustered Object external interface developed. Examples of interfaces that might be part of the external interface hierarchy include: *Counter*, *SSAC*, *Constant*, *Array*, and *HashTable*. In this way, one can create new Clustered Objects that can be used by clients expecting a specific external interface.[1]

A second hierarchy would define all the inter-representative interfaces. Examples of such interfaces are: *Invalidate*, *Update*, and *Broadcast*. These interfaces define standard methods that a representative needs to implement in order to interact in some pre-determined way.

The final interface hierarchy would define the methods that a representative implements in order to function with a Factory Object. It is expected that this hierarchy will be small, and perhaps be composed of just one standard interface.

By having separate hierarchies, an implementor can define new Clustered Objects by composing classes from the hierarchies. For example, one might want to implement a Clustered Object that supports the *Counter* external interface and has representatives that implement an invalidate protocol. This could be achieved by defining a Clustered Object whose representative class inherits both the *Counter* external interface and the *Invalidate* inter-representative interface. Of course, to function correctly with the Clustered Object's Factory object, the representative class would also have to implement a representative management interface.

The above example requires the representative class to inherit from three different interfaces. This requires language and system support for multiple inheritance. C++ does support multiple inheritance, but Tornado's dynamic type checking disallows it. In the original model, interfaces were not separated so the restriction to single inheritance was not a problem. However, the multiple interface hierarchies suggested by the new model does require multiple inheritance, which in turn

[1]It is of course possible that a new Clustered Object may implement more than one external interface.

would require a redesign of Tornado's dynamic type checking system.

7.3 Summary

We presented a new Clustered Object model, based on the experience of using the initial model and its class representation. The new model provides a more structured internal architecture of a Clustered Object. It separates out the management responsibilities into a separate Factory object. It explicitly identifies a Global Data object to explicitly manage the data that is global with respect to the representatives but local to the Clustered Object instance. Finally, the new model suggests the use of multiple inheritance to provide for more flexibility and reuse in the development of Clustered Objects.

Further work must be done to develop a class representation for the new model and to build a set of Clustered Objects with it. Options for the addition of multiple inheritance support to Tornado also need to be explored.

Chapter 8

Summary

The primary goal of Clustered Objects is to support performance optimizations typically needed in an SMP environment and at the same time support object-oriented structuring:

- Developing high performance software for CC-NUMA SMPs requires paying special attention to concurrency, the caching behaviour and the sharing and locality in memory accesses. Data structures and associated algorithms can be designed to replicate, partition and selectively place data in order to reduce sharing and maximize locality.

- Object-oriented programming uses information hiding to isolate individual data structures along with the operations on them. Externally visible operations form a strict interface for clients of a data structure and hide the internal implementation.

Clustered Objects combine the ability to replicate and partition data with information hiding. Each Clustered Object provides a well-defined external interface. Internally, however, the Clustered Object is made up of multiple representative objects that are instantiated on a per-processor basis. Each representative object supports the external interface of its Clustered Object and is associated with a specific subset of processors. The data members for a given representative object are allocated from memory local to the processor on which it was instantiated. All invocations of the external interface on a given processor are directed to a local representative object. The multiple representative objects of a Clustered Object provide a means for replicating and partitioning the data of the Clustered Object. Some of the advantages of Clustered Objects are:

- Having an object-oriented approach to information hiding means that a system composed of Clustered Objects can be customized and refined incrementally. A given Clustered Object

in the system can be easily replaced with a new Clustered Object that supports the same external interface but provides a new internal implementation.

- The ability to replicate, partition and locally place data allows Clustered Objects to be optimized for SMP environments.

- The ability to instantiate local representative objects on first use limits the resources consumed by a Clustered Object. Representatives are only created on the processors that access the Clustered Object, rather than on all processors of the system.

We have proposed two models for the structure of a Clustered Object. The first model was developed based on the Object Translation System of the Tornado operating system. Two class hierarchies were developed to support this model. Each Clustered Object is composed of a single management object called a Misshandler and a set of representative objects. The model was used to implement the example Clustered Objects studied. A second model was developed based on the experience gained from implementing the example Clustered Objects. The second model provides a standard way of implementing internal Clustered Object global data and initialization parameters. It separates out representative management from a Clustered Object's interaction with the underlying Clustered Object system. Finally the second model defines explicit interfaces between the components of a Clustered Object. The implementation and evaluation of the second model is left as future work.

In order to gain insights into the performance of Clustered Objects, two sets of example Clustered Objects were implemented and evaluated. One set consisted of multiple implementations of a simple integer counter. The performance of a simple shared implementation under a specific access pattern was used as a reference point. A performance improvement of two orders of magnitude was realized with localized non-Clustered Object implementations over the naive shared implementation. It was found that the Clustered Object implementations were able to achieve similar performance to optimized non-Clustered Object SMP implementations.

The second set of experiments entailed three different Clustered Object implementations of a more complex SMP data structure. These three implementations were compared using three different access patterns. Each implementation performed better than the other two implementations for one of the access patterns. In each case, an order of magnitude separated the best performer from the worst. This implies that the flexibility to interchange Clustered Objects will be useful as no one implementation can perform well under all access patterns. The development of a foundation

library of Clustered Objects that provides a programmer with a range of performance options is left for future work.

This work has established that Clustered Objects promises a way of developing software that is optimized for SMP systems and yet has the software engineering advantages of object-oriented techniques. Future work must be done to further refine the Clustered Object model, explore more dynamic policies and develop a foundation library.

8.1 Future Work

More experience using Clustered Objects to implement performance critical software is required to more fully develop and evaluate Clustered Objects. A systematic re-implementation of Tornado's data structures as Clustered Objects would permit a more complete evaluation. The experience gained would be invaluable in developing a more comprehensive Clustered Object model and the building of a foundation library. A complete system implemented with Clustered Objects would also provide a test bed for a thorough performance evaluation.

Other aspects that require attention are:

- support for dynamic migration of representatives.
- support for dynamic clustering. A Clustered Object could be given the ability to change its clustering degree in response to the access pattern.
- support for more variability in the representatives of a Clustered Object. For instance, different representative classes could be used within one Clustered Object.

It would also be interesting to carefully compare and evaluate the behaviour of Clustered Objects with other partitioned object models. Unfortunately at this point, no performance details for either the Fragmented Object model [22, 5] or the Distributed Object [38, 14] model have been published.

Bibliography

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 1–14, New York, October5–8 1997. ACM Press.

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

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

[4] G. Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company Inc., 2nd edition, 1994.

[5] Georges Brun-Cottan and Mesaac Makpangou. Adaptable replicated objects in distributed environments. Technical Report BROADCAST#TR95-100, ESPRIT Basic Research Project BROADCAST, June 1995.

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

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 12–25, December 1995.

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

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

[10] B. Gamsa, O. Krieger, E. Parsons, and M. Stumm. Performace issues for multiprocessor operating systems. Unpublished, University of Toronto, 1996.

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

[12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating systems. Submitted to 3rd Symposium on Operating Systems Design and Implementation, February 22-25, 1999, New Orleans, LA.

[13] Annop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of the performance of parallel applications. *1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems ACM SIGMETRICS Performance Evaluation Review*, 19(1), May 21-24, 1991.

[14] P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and W. de Jonge. An object model for flexible distributed systems. In *First Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, May 1995. http://www.cs.vu.nl/~steen/globe/publications.html.

[15] D. R. Kaeli, L. L. Fong, R. C. Booth, K. C. Imming, and J. P. Weigel. Performance analysis on a CC-NUMA prototype. *IBM Journal of Research and Development*, 41(3):205, 1997.

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

[17] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH.

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

[19] James Laudon and Daniel Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New YOrk, June2–4 1997. ACM Press.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In Jean-Loup Baer and Larry Snyder, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, June 1990. IEEE Computer Society Press.

[21] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

[22] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M Shapiro. Fragmented objects for distributed abstractions. In Thoman L. Casavant and Mukesh Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, Los Alamitos, California, 1994.

[23] Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95*, pages 480–485, Los Alamitos, CA, USA, April 1995. IEEE Computer Society Press.

[24] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[25] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 295–305, Berkeley, CA, USA, Winter 1993. USENIX.

[26] *MIPS R4000 Microprocessor User's Manual.*

[27] 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*, pages 72–81, 1995.

[28] J. Kent Peacock. File system multithreading in system v release 4 mp. In *Usenix Conference Proceedings*, pages 19–29. USENIX, 1992.

[29] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from multithreading system v release 4. In *Proceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pages 77–91. USENIX, March 1992.

[30] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[31] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[32] *White Paper: Sequent's NUMA-Q Architecture.*

[33] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[34] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[35] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and Michael Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write

network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[36] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 162–174, New York, NY, September 1992. ACM Press.

[37] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1-2):105–134, ???? 1995.

[38] M. van Steen, P. Homburg, and A. S. Tanenbaum. The architectural design of globe: A wide-area distributed sytem. Technical Report IR-442, vrige Universiteit, March 1997.

[39] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory muliprocessors. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, page 26, Pacific Grove, CA, October 1991. Published as Proc. Thirteenth ACM Symp. on Operating System Principles, volume 25, number 5.

[40] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosemblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, Cambridge, Massachusetts, 1–5 October 1996. ACM Press.

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

[42] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *Computer*, 24(1):72–79, January 1991.

[43] C. Xia and J. Torrellas. Improving the performance of the data memory hierarchy for multiprocessor operating systems. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.