(De-)Clustering Objects for Multiprocessor System Software

Eric Parsons, Ben Gamsa, Orran Krieger, Michael Stumm

Technical Report CSRI-338
November 1995

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Appeared in IWOOOS '95, pp. 72–81.

(De-)Clustering Objects for Multiprocessor System Software

Eric Parsons, Ben Gamsa, Orran Krieger, Michael Stumm
University of Toronto

{eparsons,ben}@cs.toronto.edu, {okrieg,stumm}@eecg.toronto.edu

Abstract

Designing system software for large-scale shared-memory multiprocessors is challenging because of the level of performance demanded by the application workload and the distributed nature of the system. Adopting an object-oriented approach for our system, we have developed a framework for de-clustering objects, where each object may migrate, replicate, and distribute all or part of its data across the system memory using the policies that will best meet the locality requirements for that data. The mechanism for object invocation hides the internal structure of an object, allowing a request to be made directly to the most suitable part of the object on a per-processor basis without any knowledge of how the object is de-clustered. Method invocation is very efficient, both within and across address spaces, involving no remote memory accesses in the common case. We describe the design and implementation of this framework in Tornado, our multiprocessor operating system.

1 Introduction

There are many factors to consider when writing software for modern large-scale cache-coherent multiprocessors. Because of their size, these systems have multiple, physically distributed clusters of memory and processors, connected by a high-performance inter-connection network [LLG+92, CKA91, FRB93, V+95, VSWL91]. This increases the total aggregate bandwidth of the system by replicating the memory modules and busses, but introduces the problem of high latencies for remote memory accesses due to the physical distances that must be traversed and the contention that can be experienced in the interconnection network. To obtain good performance, data must be kept physically near to the processors accessing it. Cache coherence partially addresses this issue by caching frequently accessed data, but is only effective for data which exhibits a high degree of spatial and temporal locality. Traditionally, operating systems have been shown to have very poor locality in comparison to application software [CB93].

To achieve the best possible performance, system data must be distributed, or *de-clustered*, across the system memory so as to minimize both contention and the number of remote memory references. De-clustering has been studied in isolation for many problems in operating systems, such as synchronization, memory allocation, and scheduling [MCS91, MS93, ALL89]. As part of the development of the Tornado multiprocessor operating system we wanted to provide a general-purpose framework that facilitates de-clustering, one that would encourage its use in all layers of our system.

In a previous generation of our operating system, a structuring technique called *hierarchical clustering* was developed [UKGS95]. The system was designed to manage physically proximate resources in a tightly coupled fashion, and more distantly-connected resources in a gradually more loosely-coupled fashion. Experience with this previous operating system showed that the application of this technique can result in good scalable performance for multiprocessor systems. However, as it was designed, all components of the system were required to share the same clustering hierarchy and knowledge of this hierarchy was embedded in all parts of the system. As such, hierarchical clustering failed to provide the transparency and flexibility necessary to support varying degrees of de-clustering, features that we feel are vital to achieving the best performance.

In order to improve software modularity, the general trend in operating system development, as in application software, has been towards object orientation. In developing Tornado, we wanted to incorporate the implementation-hiding aspects of object-oriented systems with the locality-enhancing aspects of hierarchical clustering, so that structural decisions applied to a single component of the system (such as whether or not to replicate some objects) could be made independently of decisions for other components of the system. This led to a model where a reference to an object (such as a page-cache) might actually refer to just one component (such as those pages cached in a nearby memory module) of a larger distributed object. Although superficially similar approaches have been applied in distributed systems through the notion of proxy objects [Sha86], a shared-memory multiprocessor, with its

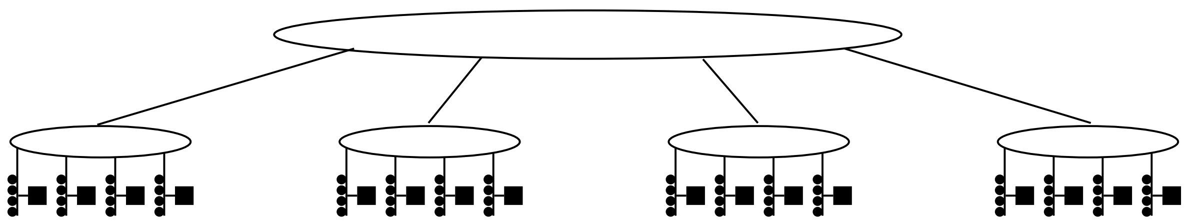

Figure 1: *In the NUMAchine architecture, nodes consist of four processors and up to 512MB of memory, and are connected using a hierarchy of rings. Our prototype, depicted in this figure, will use two levels of rings to connect 64 processors (16 nodes).*

high-degree of coupling, demands a more performance-conscious approach.

The basic model of our system is an object-oriented system, similar to Spring [HPM93, HK93], Mach-US [SJ94], and other object-oriented operating systems [Yok92, DdBF$^+$94, Dru93], in which a client request is directed to an object (e.g., a process object, memory region object, or file object) rather than to a server. We extend this basic model for a multiprocessor environment by allowing a single object reference to point to different objects on different processors, thus efficiently supporting replication, migration, and distribution of objects behind a uniform object-oriented interface. This allows the hierarchical clustering techniques explored in our previous operating system to be individually applied and tailored to each component of our new system. This increases flexibility and improves performance by facilitating a closer match between the characteristics of a service and its method of implementation.

The paper is organized as follows. First we briefly describe the operating system and hardware that are currently under development at the University of Toronto, and then discuss the problems that motivated our clustered objects. The remainder of this paper describes the mechanisms developed for clustered object invocation within as well as across address spaces. These mechanisms result in very low overhead and impose no constraints on the scalability of our system.

2 Environment

Tornado is a micro-kernel based object-oriented operating system for NUMAchine, a large-scale, cache-coherent, shared-memory multiprocessor [V$^+$95]. NUMAchine is built from nodes consisting of four processors and memory, connected by a hierarchy of high-speed, bit-parallel rings. The system is designed to be expandable to 1024 processors. Until now, we have been running the operating system on a hardware simulator and are now ready to port it to a prototype four-processor node. The final prototype system, targeted for completion in 1996, will consist of 64 R10000 MIPS processors (16 nodes) connected by a two-level hierarchy of rings, with a branching factor of four at each level (see Figure 1).

The system is expected to support a wide variety of applications, ranging from sequential interactive applications to very large parallel ones, and from scientific applications to transactional database systems. These applications can have very different requirements with respect to the way physical resources (e.g., processors, memory, and I/O) are managed. It is unlikely that a single policy for any of these resources will be able to deliver the performance demanded by all applications. In the case of memory management, for instance, some applications will perform best with large pages statically distributed among the memory modules, while others will perform best with small pages that migrate automatically. As such, the primary design goal of Tornado is to provide the flexibility needed to maximize performance, allowing applications to select, configure, or redefine their resource management objects in the most effective way [KGPS95]. This need to support a wide range of policies demands a highly flexible internal structure to the system.

3 Motivation

In a large-scale multiprocessor system, such as NUMAchine, a number of issues related to performance arise that are either particular to, or more pronounced in, this type of system. First, remote memory references have much greater latency than local memory references, leading to longer processor stall times. Second, the potential for memory and network contention increases as more processors concurrently access shared data. In a hierarchical ring structure, for example, failure to localize memory accesses within lower level rings can saturate the higher level rings. Third, blind reliance on cache-coherence to solve all locality issues can result in excessive cache-coherence network traffic and congestion at the cache controllers. Finally, false sharing at the cache-line level can compound problems in all parts of

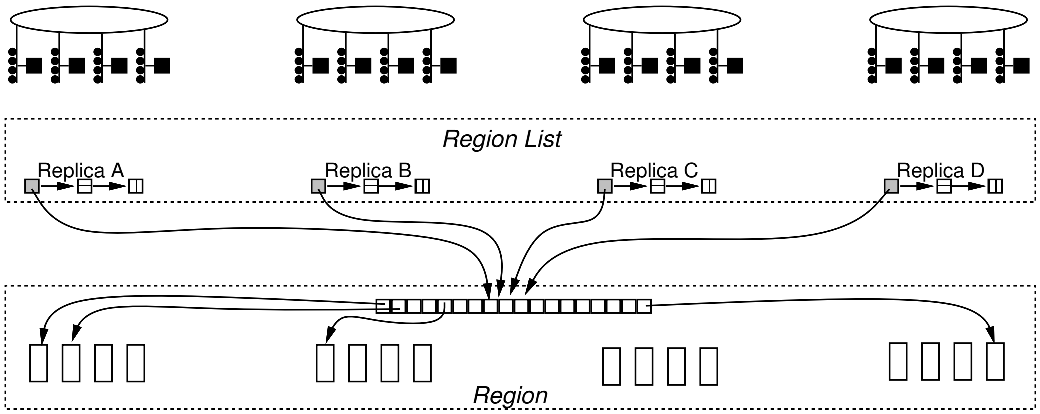

Figure 2: *This figure depicts the relationship between a region list and an individual region. The region list is replicated onto each of the lower-level rings in the system, while the region itself is partitioned across all nodes. In this figure, we illustrate a potential approach where all references to a region refer to a single object that employs a redirection table to forward requests to the component of the object local to the requesting processor. In this case, the table is a centralized resource that leads to remote memory references on each method invocation, and is therefore a potential bottleneck.*

the system, even with good cache locality.

A classic example that illustrates some of these problems in large multiprocessor systems is the process dispatch queue. Consider the case where a single linked list is used to enqueue and dequeue runnable processes, and whose head and tail pointers are stored in one memory module. The number of updates increases in proportion to the number of processors. If the pointers are left uncached, the memory module will quickly become a bottleneck, but if they are cached, coherency traffic will lead to network congestion and will likely make things worse. Dealing with such locality issues often involves certain tradeoffs, since potential solutions may entail changes to the semantics of the object. In this case, if the dispatch queue were partitioned among the processors of the system, it may no longer be possible to respect system-wide priority requirements.

Most of the solutions to these problems fall into three broad categories:

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

- An object can be replicated across the system, relying on software-based update or invalidate protocols to ensure consistency when it is modified. This approach is attractive for read-mostly objects, as the locality benefits obtained by having multiple replicas outweigh the costs of creating and managing them.

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

Each object in the system will, in general, require a different combination of techniques. For example, although the dispatch queue might be partitioned across the processors, a list of memory regions in an address space might be replicated among a cluster of processors, since it must be searched frequently (for each page fault) but changes relatively infrequently. To further complicate matters, although the list of memory regions associated with the address space might be replicated, the region objects themselves probably require a different strategy, because the contents of each individual region will change frequently as pages are brought in and ejected with each page fault. For the memory region objects, the best strategy will depend on each region's access pattern, such as whether it is accessed by a single processor or by a group of processors, and in the latter case, whether each processor accesses a private part of the region or all processors share the region in its entirety.

While hiding the internal structures and strategies of an object is important, it is also important for good performance to allow clients to directly access localized portions of the objects. Returning to the region example, the most natural way to replicate the region list is to replicate a list of object references. If a given region object has decided to partition itself in order to keep information about each page close to the processor accessing it, the question is what the region reference should refer to on two different processors.

One approach is to have both references refer to the same object and leave it to the object to determine the correct part of itself to use given the processor from which the request is occurring. However, this immediately centralizes the data structure pointed to by the common reference, which may itself become a bottleneck and eliminate any benefits realized from partitioning the region in the first place (see Figure 2). Ideally, we would like the appropriate region component to be selected automatically based on the processor that is referencing the object. The next section describes how the clustered object mechanism achieves this by distributing the redirection tables for all objects across all processors.

4 Clustered Objects

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

In the clustered object model, an object creates representative objects to which accesses may be directed. Typically, these representatives would be distributed across the system, each providing a (local) point of access for a nearby set of processors (hence called local representatives, or local reps). To reduce the object creation cost in large systems, a clustered object may specify in a lazy fashion the local rep to which method invocations from each processor should be directed. Although it is the full collection of reps that together define the implementation of the clustered object, each local rep supports the full object interface, and thus appears to client objects as the complete clustered object. The clustering structure of an object is thus completely hidden from a client, which accesses the clustered object using a single object identifier (OID).

As a simple example of a clustered object, consider an object, such as a process descriptor, which contains some long-lived read-mostly data and some data that is primarily accessed by a single processor at any given time. A possible clustering approach would be to have the read-mostly data replicated on demand and to have a single copy for the rest of the data which is migrated as required (see Figure 3). When a client accesses the object, the OID is translated into a pointer to the local rep for the processor, which in this case is the replicated data portion. If the object accesses the other part of the data (part B in the figure), then the local rep must find the location of this data to complete the operation.

In Tornado, the degree of de-clustering varies greatly among objects. For example, a program object is used to manage an address space, and in particular, is responsible for handling page faults. Since it is often the case that a program object is frequently accessed but infrequently modified, a natural approach is to create a single rep on each lower-level ring on which a process of the program is executing. If the program is a system server, it may be the case that the program's region list is updated frequently as processes are created and destroyed to handle server requests (this is in contrast to the previous example which is more characteristic of parallel scientific programs). For this reason, a region list would have a local rep for each processor whose entries are replicated on demand. Finally, some region objects, such as process stacks, will not be de-clustered at all while other region objects, such as ones used for large shared matrices, will be highly de-clustered.

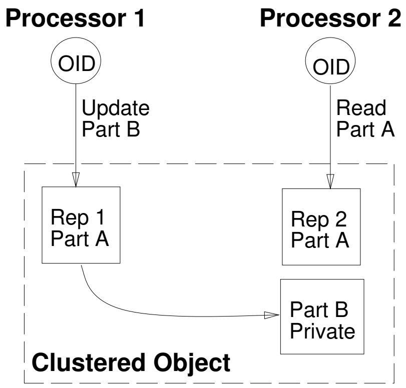

Figure 3: *This is an example of a clustered object that has one part that is frequently read (A), and another which is frequently modified (B). The object has two reps, one on each processor, to which local requests are directed. These reps contain copies of part A, thus localizing access to this part. If an operation on the object by processor 1 involves modifying part B, then the local rep must locate and modify the single copy of part B, using the necessary synchronization mechanisms. The clustered object is identified by a single OID which is valid (in this address space) across the system and used by processors 1 and 2 in this example.*

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

4.1 Object References and Translation

Although conceptually similar to a C++ reference, a clustered object identifier (OID) differs in that the object to

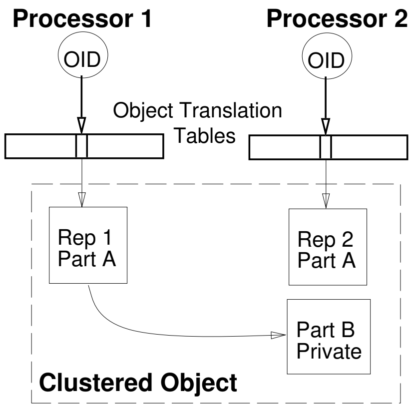

Figure 4: *Each processor has a private object translation table which is used to dereference an OID to the local rep for the processor.*

which it points depends on which processor is accessing it. It is important that invoking operations on an object be scalable and efficient, accessing only local memory in the common case, so as not to limit the potential performance of the clustered object.

For this purpose, we introduce a per-processor object translation table, which identifies the local reps that should be used for each processor. The clustered object in the previous figure would have the table entry corresponding to the object's OID on processor 1 point to rep 1, while on processor 2, point to rep 2 (see Figure 4). To invoke a method on the clustered object, the OID is used to index into the table and read the location of the local rep.

Since each processor may have a different set of local reps, all tables are mapped to the same virtual address range on each processor but backed by different physical memory. We partition the range of OIDs so that each processor can allocate new OIDs from its own subrange without having to synchronize with other processors. This enables us to reuse recently-released entries thus improving spatial locality by having multiple valid entries on a cache line, and temporal locality by reusing the same cache lines. An added benefit of this partitioning is that the root processor of a clustered object can be identified given its OID, which remains the same for the lifetime of the object.

In our implementation, an object identifier is a pointer to an entry in the object translation table, and we define all clustered objects to use C++ virtual functions[1]. The cost of invoking a clustered object method in the common case is one instruction more than for a regular C++ virtual method (6 instructions instead of 5)[2] and does not involve any remote memory references. We consider this cost to be negligible, given the locality we can obtain by having the extra level of indirection.

4.2 Translation Misses

In general, it is not possible to know *a priori* which processors will access a clustered object. For this reason, translation tables are filled and representatives are created on demand, that is, when a processor invokes an operation on a clustered object for the first time. Since objects can have different policies for creating and sharing reps, it is necessary to invoke object-specific code when an uninitialized translation table entry is accessed. While translation misses are much less frequent than hits, it is still important that both the task of directing the miss to the object specific code and the task of handing the miss are efficient and scalable.

All uninitialized entries in the translation table are set to a special null value (described later). When a translation miss occurs on such an entry, a per-processor general miss-handling object (G-MHO) is invoked which is responsible for locating and calling an object-specific miss-handling object (O-MHO) associated with the target clustered object. This O-MHO is called in the context of the processor undergoing the fault, so handling the translation miss does not interrupt any other processors unless the O-MHO does so explicitly. Typically, the O-MHO will create or locate a local rep for the processor, and fill in the appropriate translation table entry accordingly. It then forwards the call to the newly-installed local rep, which proceeds as if it had been called directly. The O-MHO may also choose not to create a local rep and handle the call itself, for example if no further calls are expected (e.g., delete object) or if the clustered object only creates local reps after a certain number of accesses have been made.

An auxiliary translation miss table is used to record the O-MHOs for all valid clustered objects. This table is partitioned across the system so that each processor maintains the portion corresponding to its range of OIDs. When a clustered object is created, it registers an O-MHO in the translation miss table. Upon a translation miss, the G-MHO examines the target OID, from which it can determine the entry in the translation miss table to call the O-MHO.

Although it may be sufficient to have a translation miss table that is partitioned (and not replicated), creating all local reps from a common (root) object will likely lead to contention. To deal with this problem, we support a generic

[1] The main reason for using virtual functions is that the type of the clustered object is not known when making cross-address-space calls, and may even be different on different processors. A second reason is that it simplifies the code path when accessing an entry that has not yet been filled on a processor (see Section 4.2).

[2] With most superscalar processors, such as the MIPS R10000, it would be possible to modify the compiler slightly so that the extra instruction is taken out of the critical path. This would result in a cycle count that is close to that of a regular virtual method call.

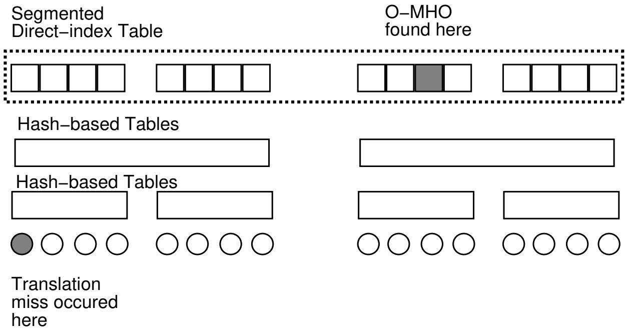

Figure 5: *This figure illustrates the hierarchy of global search tables. The top table is logically a single table but physically distributed across the system.*

way to cache the location of nearby local reps of a clustered object that might be used in handling translation misses. As illustrated in Figure 5, a hierarchy of hash tables are used to record the location of local reps that have been created in a subtree of processors below[3]. When the O-MHO of a clustered object is invoked, it can scan the hierarchy of hash tables to find the closest local rep. Sometimes, a new rep can be created directly from the one just found; if not, then the local rep may contain a direct pointer to the parts of the clustered object which are needed to create a rep. With this scheme, we still make a single memory request to a central location to obtain the address of the O-MHO, but can often handle the remainder of the miss locally, assuming the code of the O-MHO is replicated.

We use a simple scheme to avoid having to explicitly check for a translation miss on every access to an object. The null entries in the object translation table are really pointers to the G-MHO for the local processor, as if it were the real clustered object. This G-MHO has as many virtual methods as any of the clustered objects. When the first access to an object on the given processor occurs, the method in this G-MHO is invoked instead of the one for the target clustered object. This code locates and calls the O-MHO as described above, which then (usually) replaces the special null value in the table with the actual location of the processor's local rep.

4.3 Page Faults for the Translation Table

The total number of clustered objects in the system will be large, growing proportionally with the number of processors in the system. On large systems, it would be impossible to allocate physical memory for the entire object translation table on every processor. We expect, however, that the actual number of objects actively being accessed on a processor will remain relatively constant irrespective of the system size. Therefore, we allocate only a small, fixed number of physical pages on each processor to back those portions of the object translation table being actively used by that processor[4].

When an OID is dereferenced and the page containing the desired table entry is not currently mapped in, a page fault occurs. The page-fault handling code selects a victim physical page from the object translation table and writes it in compressed form to a separate fixed-size compression table. The physical page just freed is then filled with appropriate entries from the compression table. Finally, the page is mapped at the appropriate virtual address and the faulting process is restarted. Apart from some reference counts which we describe later, translation table entries do not contain any state that must be preserved. Therefore, when the compression table fills up, we can delete entries having zero reference counts, using whatever victim selection scheme is most appropriate.

As shown in Figure 6, the compressed form is simply a list of entry numbers and translation entry pairs. Since pages become progressively more sparse as the distance to objects increases, we expect to achieve good compression ratios for large parts of the table using this technique. Rather than discarding entries when the compression table overflows, we could have used paging to avoid having a limit on the size of the compression table. Typically, however, creating or finding the local rep will be less expensive than restoring the entries from disk. Also, the fact that an entry no longer exists in a processor's object translation table or compression table can be used as an indication that the local rep is no longer needed.

5 External Access to Clustered Objects

In the previous section we described how clustered objects are invoked within a single address space. Clustered objects are used within the Tornado kernel, the system servers, and system libraries in the application address spaces. Interactions between Tornado address spaces are typically between clustered objects in different address spaces. To make these efficient, method invocations across address spaces should be directed to local representatives just as in the within-address-space case, and should, in the common case, involve no remote memory accesses. In this section, we de-

[3] In NUMAchine, these subtrees correspond to rings, but they don't necessarily have to.

[4] The flexibility provided by our memory management system makes it possible for any server, not just the kernel, to manage page replacement for the object translation table.

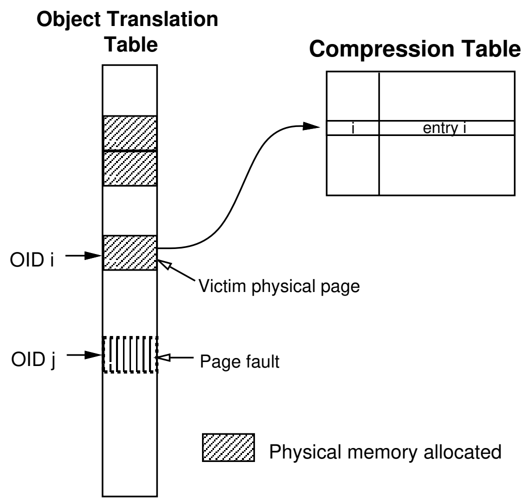

Figure 6: *This figure shows the object translation table for a processor on the left, and its compression table on the right. If there is no physical memory mapped for the entry corresponding to OID j, then an access to j will result in a page fault. A victim page is selected and its entries, such as entry i, are moved to the compression table. Subsequently, entries in the compression table corresponding to the faulting page are copied into the page, and the page is remapped at its new location.*

scribe first how objects in different address spaces interact, and how our implementation maximizes locality.

5.1 Cross-Address-Space Interactions

We differentiate between *internal* and *external* clustered objects, where only external objects can be accessed from other address spaces. This distinction is necessary for a number of reasons. First, an external object will, in general, need to authenticate the caller, while such authentication is unnecessary for within-address-space calls. Second, an external object must interface properly with the message passing facility (described in the next section). Finally, the argument conventions for external requests are slightly different with respect to pass-by-reference arguments, and the object may need to marshal and demarshal arguments.

The type of external object that we expect to be most common in Tornado is the *interface* object, used to export the interface of an internal clustered object. The interface object handles all of the server-end aspects of cross-address-space calls (authenticating the caller, de-marshalling arguments, etc.), and translates requests into corresponding requests to the internal clustered object. A clustered object might choose between several different interface implementations, each specialized for a particular type of access (e.g., read-only versus read-write).

To support cross-address-space method invocation, two additional classes of objects are generated automatically from the interface object. In the client address space, we generate a *proxy* object to interact with an interface object in the server. This allows client code to access a clustered object in a different address space as if it were a regular C++ object. The proxy object is responsible for marshalling arguments, specifying the OID of the corresponding interface object, specifying the method number, and directing the request to the appropriate server.

For every clustered object class that has external interfaces, we also generate an interface metaclass of which there is a single instance created when the server is initialized. The purpose of this metaobject is to provide an external object to which requests can be directed in order to instantiate new interface objects and to invoke static functions of the interface object class. The server ID and the OID of each metaobject, along with the type of the interface object, are recorded in a global system typeserver. It is possible for multiple servers to implement a particular interface class, in which case a request to the typeserver can optionally identify the desired one by specifying an optional server ID.

In order to create a clustered object in a server, a client performs the usual **new** operation on the proxy class. The constructor of the proxy object makes a request to the typeserver to locate the metaobject for the interface class, which it caches for future requests. It then invokes a **construct** function in the metaobject (which simply forwards the request to the corresponding static member function in the interface class) to return the OID of an interface object. The semantics of the **new** will vary from object to object. In some cases an existing interface object will be used, while in other cases a new interface object will be created, possibly also resulting in the creation of a new internal object.

5.2 Implementation Details

The basic form of interprocess communication in Tornado is the protected procedure call (PPC) [GKS94]. On startup, each Tornado server identifies to the kernel a single entry point that handles all external requests to that server. On a PPC request, proxy objects identify the target external object (i.e., interface object or metaobject) by specifying the server ID and the object's OID. The PPC facility directs the request to the specified server, where the entry point of the server performs some generic validation (described below) and makes the request to the external object.

For protection reasons, the PPC facility directs a request to a server by allocating a new process which acts on the client's behalf in the server address space. To make this efficient, the PPC facility maintains a pool of processes for

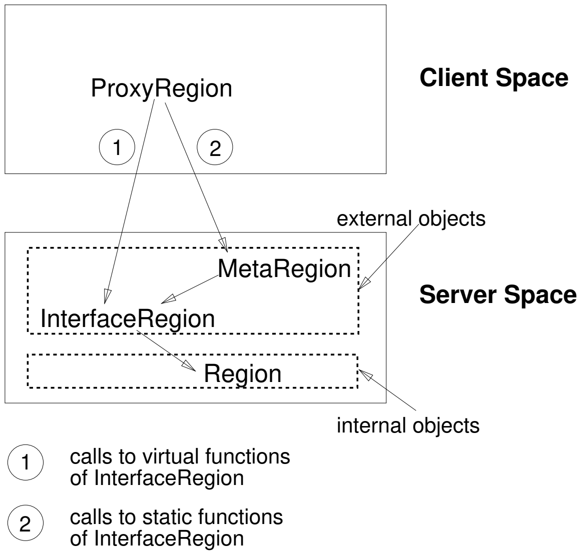

Figure 7: *This figure shows the relationship between the different objects used in client-server interactions for a generic Region object. The normal virtual method calls are handled by the interface object (InterfaceRegion) directly; the static method calls are handled by the external metaobject (MetaRegion) which simply redirects calls to static methods of the external object class.*

each server, where, in the common case, the process can be obtained from the pool. The process runs on the same processor as the client process, and handoff scheduling is used to avoid any scheduling overhead. For most requests, the arguments will fit in the registers, and these registers are passed directly from the client to the process in the server address space.

An important characteristic of the clustered object mechanism is that no remote memory accesses are necessary for object invocation when the entry in the object translation table is valid. Similarly, we would like to avoid accessing remote memory for PPC requests. To achieve this, the PPC facility employs a per-processor translation table managed in the same way as the translation table used for clustered object invocation. The entry in the translation table identified by a server's ID contains fields which identify the entry point in the server and a per-processor pool of server processes. As long the server has previously been invoked on a processor, and as long as a server process is available in the local pool, a PPC request to the server can be handled without requiring any remote memory accesses.

When an operation is invoked on a clustered object from another program, the interface object must authenticate the caller. The first time authentication occurs, the object obtains the user ID corresponding to the client program from the kernel. Once the client program has been authenticated, the interface object caches the program OID of the authenticated client, so that on subsequent calls it need only verify that the caller's OID corresponds to a program previously authenticated. (The PPC facility supplies the program OID of the client in a register on every server request, ensuring that the program OID cannot be forged and avoiding any memory accesses to obtain it.)

Some general validation must be done by the server before an external object can be invoked, and it is important that remote memory accesses are avoided during the validation. To achieve this, four additional fields exist in the per-processor clustered object translation table:

- A flag indicating whether this clustered object is externally accessible.
- A field containing the number of virtual methods defined for the clustered object. This is used to ensure that the method number passed as a parameter to the PPC call corresponds to a valid method in the external object.
- Check bits (put in unused bits in the OID) used to increase the period between reuse of an object identifier corresponding to an external object. These bits are masked off when the OID is dereferenced (recall that an OID is a pointer), but checked against the value stored in the table entry before use.
- An optional program OID field which identifies a particular client program that can make requests to the external object. If this field is valid, the PPC-handling code verifies that the OID of the calling program matches the value stored before allowing the invocation to continue.

Given an OID and a method number as an argument to a PPC call, the server entry-point code performs the following sequence of operations: it verifies that the OID is in the range of valid object identifiers, is properly aligned, and corresponds to a valid external object; it then verifies that the method number is in the correct range and, if the program OID field in the table is valid, verifies that the program OID of the caller matches the stored value; finally it dereferences the pointer to the local rep recorded in the table entry and invokes the virtual method indexed by the method number. In the common case of a one-to-one relationship between a proxy and interface object, the program OID field in the translation table is the only information needed for authentication, and hence full locality can be achieved without requiring a local rep for the interface object.

Deleting external clustered objects pose special difficulties since a method invocation from a client program can arrive at any arbitrary point. Some mechanism must be used to ensure that all current external method invocations have terminated before data is released. The object translation table contains a reference count field for this purpose which records the number of outstanding references to the clustered object. When the object is accessed externally, this value is incremented for the duration of the request. Internally, this field can be used to ensure that the clustered object is not removed unexpectedly. To prevent any further requests to an object being deleted from occurring, the external flag and check bits for the object are cleared.

Although a cross-address-space invocation may appear complicated on the surface, the cost is actually quite modest. A null cross-address-space invocation requires approximately 250 instructions, including about 30 instructions in the server for validating and authenticating the client's request.

6 Conclusions

There has been a significant amount of work studying the use of object orientation in operating systems, having led to the development of software that is both flexible and modular. As such, we have designed Tornado to be highly object-oriented, using fine-grained objects as building blocks in the creation of larger objects. An underlying requirement of our system, or more generally of large-scale shared-memory multiprocessors, however, is that it also achieve a very high level of performance. The implication of the architecture of these systems is that many system software objects must be de-clustered to achieve a high level of concurrency and to make efficient use of the hardware resources.

In distributed systems, the performance and reliability benefits of replicating data are well known. Although similar in nature, large-scale multiprocessors have much smaller communication costs. As a result, the overhead of accessing clustered objects—in particular finding and accessing the part of the object to which requests should be directed—must be extremely low.

We expect to use de-clustering at all levels of the operating system, from the kernel to servers to application-level libraries. The framework we describe in this paper allows objects to do so in a transparent fashion. An object can associate a local representative with each processor which is used for local accesses and which may be changed independently of other objects. Internally, the object can use replication, migration, distribution, or any combination of these to obtain the desired level of performance.

This framework, including the tools used to generate proxy and metaclass objects, has been fully implemented and is now running as part of the Tornado operating system. Nearly all of our kernel and server objects, including programs, memory regions, process descriptors, and files are implemented as clustered objects.

References

[ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Performance Evaluation Review*, 17(1):49–60, May 1989.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth Symposium on Operating System Principles*, pages 120–133, 1993.

[CKA91] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *ASPLOS-IV Proceedings*, pages 224–234, Santa Clara, California, April 8-11 1991.

[DdBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Kenskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

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

[FRB93] S. Frank, J. Rothnie, and H. Burkhardt. The KSR1: Bridging the gap between shared memory and MPPs. In *IEEE Compcon 1993 Digest of Papers*, pages 285–294, 1993.

[GKS94] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proceedings of the 1994 International Parallel Processing Symposium*, pages 208–211, Boca Raton, FL, August 1994.

[HK93] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix Conference*, 1993.

[HPM93] Graham Hamilton, Michael L. Powell, and James J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 69–79, 1993.

[KGPS95] Orran Krieger, Ben Gamsa, Eric Parsons, and Michael Stumm. Performance-oriented flexibility using building-block compositions. Submitted for publication, 1995.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Henessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63, March 1992.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MS93] Paul E. McKenney and Jack Slingwine. Efficient kernel memory allocation on shared-memory multiprocessors. In *USENIX Technical Conference Proceedings*, pages 295–305, San Diego, CA, Winter 1993.

[Sha86] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Systems*, pages 198–204, 1986.

[SJ94] J. Mark Stevenson and Daniel P. Julin. Client-server interactions in multi-server operating systems: The Mach-US approach. Technical Report CMU-CS-94-191, Carnegie-Mellon University, September 1994.

[UKGS95] Ron Unrau, Orran Krieger, Ben Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[V+95] Z. Vranesic et al. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, April 1995.

[VSWL91] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. The Hector multiprocessor. *IEEE Computer*, 24(1), January 1991.

[Yok92] Yasuhiko Yokote. The Apertos reflective operating system — the concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–434, Vancouver, BC CD, October 1992. ACM Press, New York, NY , USA. Published as SIGPLAN Notices, volume 27, number 10.