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

Ben Gamsa* Orran Krieger† Jonathan Appavoo* Michael Stumm*

*Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{ben,jonathan,stumm}@eecg.toronto.edu

†IBM T.J. Watson Research Center
Yorktown Heights, New York
okrieg@us.ibm.com

Abstract

We describe the design and implementation of Tornado, a new operating system designed from the ground up specifically for today's shared memory multiprocessors. The need for improved locality in the operating system is growing as multiprocessor hardware evolves, increasing the costs for cache misses and sharing, and adding complications due to NUMAness. Tornado is optimized so that locality and independence in application requests for operating system services—whether from multiple sequential applications or a single parallel application—are mapped onto locality and independence in the servicing of these requests in the kernel and system servers. By contrast, previous shared memory multiprocessor operating systems all evolved from designs constructed at a time when sharing costs were low, memory latency was low and uniform, and caches were small; for these systems, concurrency was the main performance concern and locality was not an important issue.

Tornado achieves this locality by starting with an object-oriented structure, where every virtual and physical resource is represented by an independent object. Locality, as well as concurrency, is further enhanced with the introduction of three key innovations: (i) *clustered objects* that support the partitioning of contended objects across processors, (ii) a *protected procedure call* facility that preserves the locality and concurrency of IPC's, and (iii) a new locking strategy that allows all locking to be encapsulated within the objects being protected and greatly simplifies the overall locking protocols. As a result of these techniques, Tornado has far better performance characteristics, particularly for multithreaded applications, than existing commercial operating systems. Tornado has been fully implemented and runs both on Toronto's NUMAchine hardware and on the SimOS simulator.

1 Introduction

Traditional multiprocessor operating systems, including those commercially available today (e.g., IBM's AIX, Sun's Solaris, SGI's IRIX, HP's HP/UX), all evolved from designs when multiprocessors were generally small, memory latency relative to processor speeds was comparatively low, memory sharing costs were low, memory access costs were uniform, and caches and cache lines were small.[1] The primary performance concern for these systems was to maximize concurrency, primarily by identifying contended locks and breaking them up into finer-grained locks.

Modern multiprocessors, such as Stanford's Dash and Flash systems, Toronto's NUMAchine, SGI's Origin, HP/Convex Exemplar, and Sun's larger multiprocessors, introduce new serious performance problems because of (i) higher memory latencies due to faster processors and more complex controllers, (ii) large write sharing costs, (iii) large secondary caches, (iv) large cache lines that give rise to false sharing, (v) NUMA effects, and (vi) larger system sizes, that stress the bottlenecks in the system software and uncover new ones. These characteristics all require that the system software be optimized for locality, something that was not necessary in the past. In addition to maximizing temporal and spatial locality as required for uniprocessors, optimizing for locality in the context of modern multiprocessors also means:

- minimizing read/write and write sharing so as to minimize cache coherence overheads,
- minimizing false sharing, and
- minimizing the distance between the accessing processor and the target memory module (in the case of NUMA multiprocessors).

To understand the importance of locality in modern multiprocessors, consider a simple counter (for example, a performance counter or a reference count) with multiple threads concurrently updating it. Figure 1 shows the performance of different implementations of such a counter (in cycles per update) when run on SimOS simulating a 16 processor, 4 processor-per-node NUMA multiprocessor. In these experiments, sufficient delays are introduced between counter updates to ensure that the counter is not contended. The shared counter scales well if a system is simulated with hardware parameters set to those typical of ten years ago (third curve from the bottom), but it

[1] In fact, most of these systems evolved from uniprocessor operating system designs.

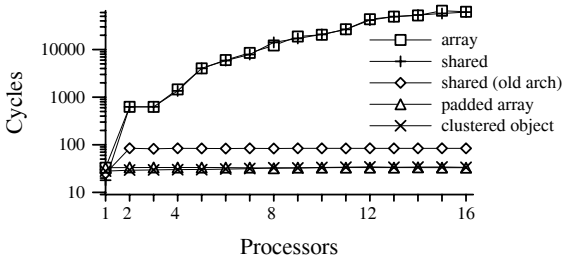

Figure 1: *Cycles per update of a counter (log scale).*

performs orders of magnitude worse when simulating a modern shared memory multiprocessor, such as NUMA-chine (top-most curve).[2] The counter performs no better when the counter is converted to an array with each processor updating its own counter individually (complicating the extraction of the total counter value) because of false sharing. Good performance can be achieved only by additionally padding each array entry to the secondary cache line size, or by applying the clustered object techniques described later in this paper (bottom two curves).

In general, data structures that may have been efficient in previous systems, and might even possess high levels of concurrency, are often inappropriate in modern systems with high cache miss and write sharing costs. Moreover, as the counter example above demonstrates, a single poorly constructed component accessed in a critical path can have a serious performance impact. While the importance of locality has been recognized by many implementors of shared memory multiprocessor operating systems, it can be extremely difficult to retrofit locality into existing operating system structures. The partitioning, distribution, and replication of data structures as well as the algorithmic changes needed to improve locality are difficult to isolate and implement in a modular fashion, given a traditional operating system structure.

The fact that existing operating system structures have performance problems, especially when supporting parallel applications, is exemplified in Figure 2, which shows the results of a few simple micro-benchmarks run on a number of commercial multiprocessor operating systems.[3] For each commercial operating system considered, there is a significant slowdown when simple operations are issued in parallel that should be serviceable completely independently of each other.

In this paper, we describe the design and implementation of Tornado, a new shared memory multiprocessor operating system that was designed from the ground up 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 system kernel and servers. More specifically, Tornado is designed to service all OS requests on the same processor they are issued on, and to handle requests to different resources without accessing any common data structures and without acquiring any common locks. As a result, Tornado does not exhibit the difficulties of the aforementioned systems (see Figure 2). Moreover, we found that we could achieve this 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.

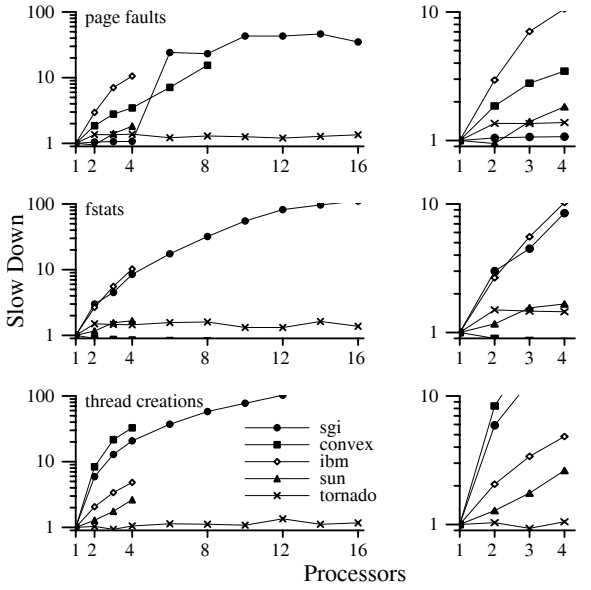

Figure 2: *Normalized cost (log scale) of simultaneously performing on n processors: n in-core page faults (top), n fstats (middle), and n thread creations/deletions (bottom) for 5 commercial shared memory multiprocessor operations systems and for Tornado. A full description of these experiments can be found in Section 7.*

Tornado uses an object-oriented approach, where every virtual and physical resource in the system is represented by an independent object, ensuring natural locality and independence for all resources. Aside from its object-oriented structure, Tornado has three additional innovations that help maximize locality (as well as concurrency). First, *Clustered Objects* allow an object to be partitioned into *representative objects*, where independent requests on different processors are handled by different representatives of the object in the common case. Thus, simultaneous requests from a parallel application to a single virtual resource (i.e., page faults to different pages of the same memory region) can be handled efficiently preserving as much locality as possible.

Second, the Tornado *Protected Procedure Call* facility maps the locality and concurrency in client requests to the servicing of these requests in the kernel and sys-

[2] Also, although not shown, the number of cycles required per update varies greatly for the different threads due to NUMA effects.

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

tem servers, and yet performs competitively with the best uniprocessor IPC facilities. Thus, repeated requests to the same server object (such as a read for a file) are serviced on the same processor as the client thread, and concurrent requests are automatically serviced by different server threads without any need for data sharing or synchronization to start the server threads.

Finally, Tornado uses a semi-automatic garbage collection scheme that facilitates localizing lock accesses and greatly simplifies locking protocols. As a matter of principle, all locks are internal to the objects (or more precisely their representatives) they are protecting, and no global locks are used. In conjunction with clustered object structures, the contention on a lock is thus bounded by the clients of the representative being protected by the lock. With the garbage collection scheme, no additional (existence) locks are needed to protect the locks internal to the objects. As a result, Tornado's *locking strategy* results in much lower locking overhead, simpler locking protocols, and can often eliminate the need to worry about lock hierarchies.

The foundation of the system architecture is Tornado's object-oriented design strategy, described in Section 2. This is followed by a description of the three key components discussed above: clustered objects (Section 3), locking (Section 5), and protected procedure calls (Section 6), with a short interlude to consider memory allocation issues in Section 4. Although we focus primarily on the Tornado kernel, it is important to note that these components are also used in the implementation of the Tornado system servers.

Tornado is fully implemented (in C++), and runs on our 16 processor NUMAchine [14, 31] and on the SimOS simulator [27]; it supports most of the facilities (e.g., shells, compilers, editors) and services (pipes, TCP/IP, NFS, file system) one expects. Experimental results that demonstrate the performance benefits of our design are presented in Section 7, followed by an examination of related work in Section 8 and concluding remarks in Section 9.

2 Object-oriented structure

Operating systems are driven by the requests of applications on virtual resources such as virtual memory regions, network connections, threads, address spaces, and files. To achieve good performance on a multiprocessor, requests to different virtual resources should be handled independently; that is, without accessing any shared data structures and without acquiring any shared locks. One natural way to accomplish this is to use an object-oriented strategy, where each resource is represented by a different object in the operating system.

As an example, Figure 3 shows, in a slightly simplified form, the key objects used for memory management in Tornado. On a page-fault, the exception is delivered to the *Process* object for the thread that faulted. The *Process* object maintains the list of mapped memory regions in the process's address space, which it searches to identify the responsible *Region* object to forward the request to. The region translates the fault address into a file offset, and forwards the request to the *File Cache Manager* (FCM) for the file backing that portion of the address space. The FCM checks if the file data is currently cached in memory. If it is, then the address of the corresponding physical page frame is returned to the *Region*, which makes a call to the *Hardware Address Translation* (HAT) object to map the page, and then returns. Otherwise, the FCM requests a new physical page frame from the DRAM manager, and asks the *Cached Object Representative* (COR) to fill the page from a file. The COR then makes an upcall to the corresponding file server to read in the file block. The thread is re-started when the file server returns with the required data.

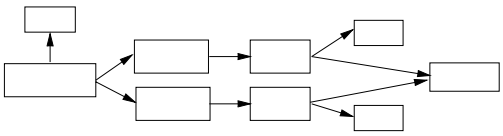

Figure 3: *The key memory management object relationships in Tornado.*

This example illustrates the advantage of employing an object-oriented approach. In the performance critical case of an in-core page fault, all objects invoked are specific to either the faulting process or the file(s) backing that process; the locks acquired and data structures accessed are internal to the objects. Hence, when different processes are backed by (logically) different files, there are no potential sources of contention. Also, for processes running on different processors, the operating system will not incur any communication misses when handling their faults. In contrast, many operating systems maintain a global page cache or a single HAT layer which can be a source of contention and offers no locality.

Localizing data structures in the Tornado fashion results in some new implementation and policy tradeoffs. For example, without a global page cache, it is difficult to implement global policies like a clock replacement algorithm in its purest form. Memory management in Tornado is based on a working set policy (similar to that employed by NT [10]), and most decisions can be made local to FCMs.

In Tornado, most operating system objects have multiple implementations, and the client or system can choose the best implementation to use at run time. The implementation can be specific to the degree of sharing, so implementing an object with locking protocols and data structures that scale is only necessary if the object is widely shared. As a result, a lower overhead implementation can be used when scalability is not required. We have found the object-oriented structure of Tornado to greatly simplify its implementation, allowing us to initially implement services using only simple objects with limited

concurrency, improving the implementation only when performance (or publication) required it. In the future we expect to be able to dynamically change the objects used for a resource.

Although our object-oriented structure is not the only way to get the benefits of locality and concurrency, it is a natural structuring technique and provides the foundation for other Tornado features such as the clustered object system [24] and the building block system [1].

3 Clustered Objects

Although the object-oriented structure of Tornado can help reduce contention and increase locality by mapping independent resources to independent objects, some components, such as a File Cache Manager (FCM) for a shared file, a Process object for a parallel program, or the system DRAM Manager, may be widely shared and hence require additional locality enhancing measures.

Common techniques used to reduce contention for heavily shared components include replication, distribution, and partitioning of objects. For example, for the (performance) counter discussed in the introduction, full distribution of the counter is used to ensure each processor can independently update its component, at the cost of making the computation of the true current value (i.e., the sum of all elements) more complicated. As another example, consider the thread dispatch queue. If a single list is used in a large multiprocessor system, it may well become a bottleneck and contribute significant additional cache coherency traffic. By partitioning the dispatch queue so that each processor has a private list, contention is eliminated.[4]

These types of optimizations have been applied before in other systems, but generally in an ad hoc manner, and only to a few individual components. The goal of Tornado's clustered object system is to facilitate the application of these techniques as an integral part of a system's overall design [24].

3.1 Overview

A clustered object presents the illusion of a single object, but is actually composed of multiple component objects, each of which handles calls from a specified subset of the processors (see Figure 4). Each component object represents the collective whole for some set of processors, and is thus termed a clustered object *representative*, or just *rep* for short. All clients access a clustered object using a common clustered object reference (that logically refers to the whole), with each call to the clustered object automatically directed to the appropriate local representative.

The internal structure of a clustered object can be defined in a variety of ways. One aspect of this variety

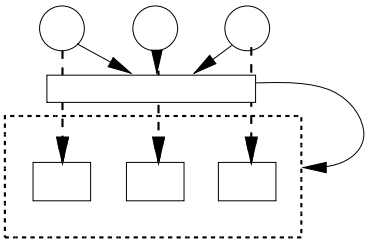

Figure 4: *An abstract depiction of a clustered object.*

is the *degree of clustering*. There might be one rep for the entire system, one rep per processor, or one rep for a cluster of neighboring processors. For example, in the case of the memory management subsystem of Tornado, the Cached Object Representative (COR) has a single rep that is shared across all processors (thus acting as a regular shared object) since it is read-mostly and only invoked when file I/O is required (which occurs relatively infrequently). The Region is also read-mostly, but is on the critical path for all page faults, and hence can benefit from partial replication, with one rep per cluster of processors. On the other hand, the FCM maintains the state of the pages of a file cached in memory, and hence can benefit from a partitioning strategy, where the hash table for the cache is distributed across a number of reps (at least for files that are being widely shared).

With multiple reps per object, it is necessary to keep them consistent. A variety of strategies are possible, including invalidation and update protocols. Coordination between reps of a given object can occur either through shared memory or a form of remote execution in Tornado called remote PPCs (described later in Section 6). Although shared memory is generally more efficient for fine-grained operations, it can sometimes be cheaper to incur the cost of the remote execution facility and perform the operation local to the data, if there is the possibility of high contention, or if there is a large amount of data to be accessed. With an efficient remote execution facility, the tradeoff point can be as low as a few tens of cache misses.

The use of clustered objects has several benefits. First, it facilitates the types of optimizations commonly applied on multiprocessors, such as replication or partitioning of data structures and locks. Second, it preserves the strong interfaces of object-oriented design so that clients need not concern themselves with the location or organization of the reps, and just use clustered object references like any other object reference. All complexity resides in the internal implementation of the clustered objects and the clustered object system.

Third, clustered objects enable incremental optimizations. Initially, a clustered object might be implemented with just one rep serving requests from all processors. This implementation would be almost identical to a corresponding non-clustered implementation. If performance requirements demand it, then the clustered object can be

[4] However, it too has complications, due to the difficulty of ensuring good load balancing and respecting system-wide priorities.

successively optimized independently of the rest of the system.

Fourth, several different implementations of a clustered object may exist, each with the same external interface, but each optimized for different usage patterns.

Finally, the clustered object system supports dynamic strategies where the specific type of representative can be changed at run time based on the type and distribution of requests.

3.2 Application of Clustered Objects

In our implementation, large-grain objects, like FCM, Region, and Thread objects, are candidates for clustered objects, rather than smaller objects like linked lists. One example that illustrates many of the benefits of our approach is the Process object.

Because a Process can have multiple threads running on multiple processors and most of the Process object accesses are read-only, the Process clustered object is replicated to each processor the process has threads running on. On other processors, a simple rep is used that redirects all calls to one of the full reps. Some fields, like the base priority, are updated by sending all modifications to the home rep and broadcasting an update to all the other reps. Other components, like the list of memory Regions in the process, are updated on demand as each rep references the Region for the first time, reducing the cost of address space changes when Regions are not widely shared. If threads of the program are migrated, the corresponding reps of the Process object are migrated with them to maintain locality.

As an illustration of some of the tradeoffs with clustered objects, consider Figure 5(a) which shows the performance of page fault handling for a multithreaded program (more details on the experiments are provided in Section 7). With multiple reps for the Process object, the Region list is replicated and page faults can be processed with full concurrency, compared to the simple shared case where a lock on the Region list creates a bottleneck. The clustered object structure effectively splits the Process lock among the representatives, allowing each rep to lock its copy independently, thus eliminating both the write-sharing of the lock and its contention. Although it would appear that a similar effect could be achieved with a reader-writer lock protecting a single shared Process object, the lock is normally held for such a short duration that the overheads of the reader-writer lock (including the write-sharing it entails) would overshadow any concurrency benefits.

However, by replicating the Process object, other operations, such as deleting a memory Region, become more expensive (see Figure 5(b)) because of the need to keep the Process object reps consistent. This tradeoff is generally worthwhile for the Process object, since page faults are much more common than adding or deleting regions.

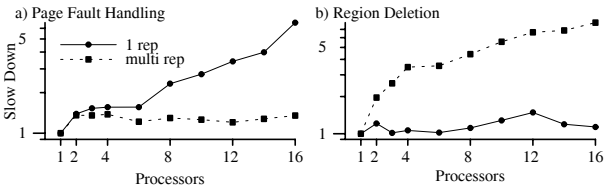

Figure 5: *Comparison of the performance of a Process clustered object with one rep vs. a Process clustered object with n reps (one per processor) for (a) in-core page fault handling and (b) Region destruction*

3.3 Clustered Object implementation

The key to the implementation of clustered objects is the use of per-processor translation tables. For each clustered object the tables maintain a pointer to the rep responsible for handling method invocations for the local processor. A clustered object reference is just a pointer into the table, with the extra level of indirection through the local table providing the mechanism for locating the local rep. (A clustered object call thus requires just one extra instruction.) By having each per-processor copy of the table located at the same virtual address, a single pointer into the table will refer to a different entry (the rep responsible for handling the invocation) on each processor.

Because it is generally unknown *a priori* which reps will be needed when and where, reps are typically created on demand when first accessed. This is accomplished by requiring each clustered object to define a miss handling object[5] and by initializing all entries in the translation tables to point to a special global miss handling object (see Figure 6). When an invocation is made for the first time on a given processor, the global miss handling object is called. Because it knows nothing about the intended target clustered object, it blindly saves all registers and calls the clustered object's miss handler to let it handle the miss. The object miss handler then, if necessary, creates a rep and installs a pointer to it in the translation table; otherwise, if a rep already exists to handle method invocations from that processor, a pointer to it is installed in the translation table. In either case, the object miss handler returns with a pointer to the rep to use for the call. The original call is then restarted by the global miss handler using the returned rep, and the call completes as normal, with the caller and callee unaware of the miss handling work that took place. This whole process requires approximately 150 instructions,[6] which although non-negligible, is still inexpensive enough to be used as a general purpose mechanism for triggering dynamic actions beyond just inserting a rep into the table.[7]

[5] Default miss handlers are provided for the common cases.

[6] All references to instructions in the paper are for MIPS instructions, which should be comparable to most other RISC processors. Results from full experimental tests are presented in Section 7.

[7] The entry can also be pre-filled to avoid the cost of the miss, should the rep structure be known in advance.

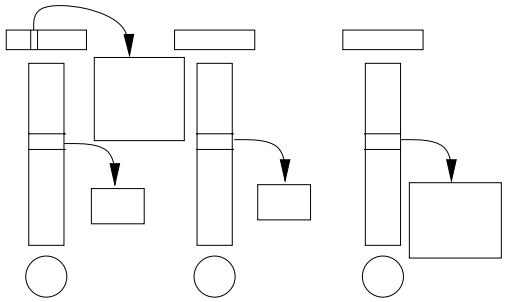

Figure 6: *Overview of clustered object implementation. Clustered object i has been accessed on P0 and P1, where reps have been installed; P2 has not yet accessed object i.*

Miss handling requires that all public methods of a clustered object be virtual and that a standard error code type be returned to allow the global miss handler to return an error when necessary. This restriction has not been a problem for us, since Tornado uses an object-oriented structure, and since most clustered objects have abstract base classes to allow different implementations.

To allow the global miss handler to locate the clustered object, the clustered object miss handler is installed at creation time in a global miss handling object table. This table is not replicated per-processor, but instead is partitioned, with each processor receiving a fraction of the total table. The global miss handler can then use the object reference that caused the miss to locate the clustered object's miss handler. This interaction with the clustered object system adds approximately 300 instructions to the overhead of object creation and destruction. Again, although significant, we feel the various benefits of clustered objects make this cost reasonable, especially if the ratio of object calls to object creation is high, as we expect it to be.

Finally, the translation tables are likely to be sparsely populated, because (i) there can be a large number of clustered objects (tens of thousands per processor), (ii) the translation table on each processor has to be large enough to handle all objects created anywhere in the system, and (iii) many clustered objects are only accessed on the processor on which they are created. As a result, the translation tables reside in virtual memory (even in the kernel), with pages only allocated when needed. However, instead of paging out pages when memory runs low, our implementation simply discards victim pages, since the table is really just a cache of entries, with the miss handlers of the clustered objects keeping track of the existence and location of the reps (i.e., they maintain the backing copy).[8]

[8] As an optimization, it would make sense to compress the victim pages to a fixed size compression table (i.e., second-level cache), since many of the pages are sparsely populated, but we have not yet implemented this.

4 Dynamic Memory Allocation

For the clustered object approach to be effective, it requires a facility that supports processor-local memory allocation. The memory allocator must be efficient, highly concurrent, and maximize locality both in its internal design and in the memory it returns on allocation.

Tornado's initial memory allocation facility used pools of memory per processor to support locality in memory allocations, using a design similar to that of [23]. However, we found that per-processor pools were not sufficient due to cache line false sharing problems that occur with small block allocations. We addressed this by providing a separate per-processor pool for small blocks that are intended to be accessed strictly locally.[9] Additionally, to address NUMA locality issues, the Tornado allocator partitions the pools of free memory into clusters. Although this requires an extra check on each free to determine the home cluster of the target block, this adds only three instructions to the critical path, most of which can be hidden in super-scalar processors. Finally, to support user-level allocations, instead of disabling interrupts, we use an optimized locking protocol that takes advantage of the common availability of load-linked/store-conditional instructions in today's processors to reduce locking overhead to just four instructions.

The allocator requires only 16 instructions for common case allocation, and 21 instructions for the common case deallocation (including checks for remote frees and overfull free lists), while providing locality, concurrency, and efficiency.[10]

5 Synchronization

There are two kinds of locking issues in most systems: those related to concurrency control with respect to modifications to data structures, which we refer to simply as *locking*, and those related to protecting the existence of the data structures; i.e., providing *existence guarantees* to ensure the data structure containing the variable is not deallocated during an update. We discuss each in turn.

5.1 Locking

One of the key problems with respect to locking is its overhead. In addition to the basic instruction overhead of locking, there is the cost of the extra cache coherence traffic due to the intrinsic write-sharing of the lock. With Tornado's object-oriented structure it is natural to encapsulate all locking within individual objects. This helps reduce the scope of the locks and hence limits contention.

[9] Another approach would been to make the minimum block size of the allocator the same as the cache line size, but at 128 bytes, this can increase internal fragmentation considerably.

[10] Although difficult to accurately measure on a super-scalar out-of-order processor, on a MIPS R10000 processor the cost of a malloc/free pair was about 7 cycles over and above that of a null function call.

Moreover, the use of clustered objects helps limit contention further by splitting single objects into multiple representatives thus limiting contention for any one lock to the number of processors sharing an individual representative. This allows us to optimize for the uncontended case. We use highly efficient spin-then-block locks, that require only two dedicated bits[11] from any word (such as the lower bits of an aligned pointer), at a total cost of 20 instructions for a lock/unlock pair in the uncontended case.[12]

5.2 Existence guarantees

Providing existence guarantees is likely the most difficult aspect of concurrency control. The traditional way of eliminating races between one thread trying to lock an object and another deallocating it, is to ensure that all references to an object are protected by their own lock, and all the references are used only while holding the lock on the reference. The disadvantage of this approach is that the reference lock in turn needs its own protector lock, with the pattern repeating itself until some root object that can never be deallocated. This results in a complex global lock hierarchy that must be strictly enforced to prevent deadlock, and it encourages holding locks for long periods of time while operations on referenced objects (and their referenced objects) are performed. For example, in the page fault example, the traditional approach would require holding a lock on the process object for the duration of the page fault, solely to preserve the continued existence of the Regions it references.[13]

For Tornado, we decided to take a somewhat different approach, using a semi-automatic garbage collection scheme for clustered objects. With this approach, a clustered object reference can be safely used at any time, whether any locks are held or not, even as the object is being deleted. This simplifies the locking protocol, often eliminating the need for a lock completely (for example, for read-only objects). It also removes the primary reason for holding locks across object invocations, increasing modularity and obviating the need for a lock hierarchy in most cases.

5.3 Garbage collection implementation

The key idea in the implementation of our semi-automatic garbage collection scheme is to distinguish between what we call temporary references and persistent references. Temporary references are all clustered object references that are held privately by a single thread, such as references on a thread's stack; these references are all implicitly destroyed when the thread terminates. In contrast, persistent references are those stored in (shared) memory; they can be accessed by multiple threads and might survive beyond the lifetime of a single thread.

The distinction between temporary and persistent references is used to divide clustered object destruction into three phases. In the first phase, the object ensures that all persistent references to it have been removed. This is part of the normal cleanup procedure required in any system; when an object is to be deleted, references to the object must be removed from lists and tables and other objects.

In the second phase, the clustered object system insures that all temporary references have been eliminated. To achieve this, we chose a scheme that is simple and efficient. It is based on the observation that the kernel and system servers are event driven and that the service times for those events are relatively short. In describing our scheme, first consider the uniprocessor case. The number of operations (i.e., calls to the server from some external client thread) currently active is maintained in a per-processor counter; the counter is incremented every time an operation is started and decremented when the operation completes. Thus, when the count is zero, we know there can be no live temporary references to any object on that processor, and phase two ends.[14] A concern with this approach is that there is no guarantee that the count of live threads will ever actually return to zero, which could lead to a form of starvation. However, since system server calls tend to be short, we do not believe this to be a problem in practice.[15]

For the multiprocessor case, we need to also consider threads running on other processors with temporary references to the object. Each clustered object can easily know which set of processors can access it, because the first access to an object on a processor always results in a translation table miss, and any reference stored in an object can be accessed only by the set of processors that have already made an access to the object. Hence, the set of processors can be determined when cleaning up the persistent references by determining which processors have objects with a persistent reference to the target clustered object and forming the union of the set. We use a circulating token scheme to determine that the per-processor counters have reached zero on each processor of the target processor set, with the token visiting each processor

[11] One bit indicates if the lock is held and the other indicates if there are queued threads. A separate shared hash table is used to record the list of waiting threads.

[12] We measured the lock time on an R10000 processor in a manner similar to memory allocation, and found it cost about 10 cycles over the cost of a pair of null function calls.

[13] A different approach altogether is to use lock-free concurrency control. However, practical algorithms often require additional instructions not currently found on modern processors, and they have their own difficulties in dealing with memory deallocation [15, 18]

[14] This is a much stronger requirement than actually required; it would, for example, suffice to ensure that all threads have completed that were active when object destruction was initiated, but that would require keeping track of much more information that just a raw count.

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

that potentially holds references to the object being destroyed. When a processor receives the token it waits until its count of active threads goes to zero, before passing it on to the next processor. When the token comes back to the initiating processor it knows that the active count has gone to zero on all processors since it last had the token.[16]

Finally, when all temporary references have been eliminated, the clustered object can be destroyed (i.e., its reps can be destroyed, their memory released, and the clustered object entry freed).

Unfortunately, we have not yet tuned this code, so it currently requires approximate 270 instructions to fully deallocate an object once it has been handed to the cleanup system.

6 Interprocess communication

In a microkernel system like Tornado that relies on client-server communication, it is important to extend the locality and concurrency of the architecture beyond the internal design of individual components to encompass the full end-to-end design. For example, in the memory management subsystem, each page fault is an implicit call from the faulting thread to the process object; each call by the Cached Object Representative (COR) to its corresponding file object in the file system is another cross-process object call; and each call from the file system to the device driver object is another. Concurrency and locality in these communications are crucial to maintaining the high performance that exists within the various Tornado subsystems and servers.

Tornado uses a Protected Procedure Call (PPC) model [13], where a call from a client object to a server object acts like a clustered object call that crosses from the protection domain of the client to that of the server, and then back on completion. The key advantages of the PPC model are that: (i) client requests are always serviced on their local processor; (ii) clients and servers share the processor in a manner similar to handoff scheduling; and (iii) there are as many threads of control in the server as client requests. This also means that any client-specific state maintained in the server can be held local to the client, reducing unnecessary cache traffic. For example, for page faults to memory-mapped files that require I/O, all of the state concerning the file that the client has mapped can be maintained in data structures local to the thread that is accessing the mapped file. In some sense, this is like extending the Unix trap-to-kernel process model to all servers (and the kernel for Tornado), but without needing to dedicate resources to each client, as all clients use the same port to communicate to a given server, including the kernel. The PPC model is thus a key component to enabling locality and concurrency within servers.

[16]To deal with scalability, there can be multiple tokens circulating, covering different subsets of processors.

To support cross-process clustered object calls, a stub generator is provided that generates stubs based on the public interface of a clustered object. The clustered object system also includes support in the way of a few extra bits in the object translation table that identify those clustered objects that can accept external calls. To ensure references invoked from an external source are valid, the PPC subsystem checks to ensure they fall within the translation table, are properly aligned, and pass the security bits check. This makes it easy to use clustered object references, in conjunction with the identity of the target server, as a global object reference. As a result, cross-process clustered object calls are made directly to the local rep of the target object, providing the same locality and concurrency benefits for cross-process calls as for in-process calls.

The PPC facility also supports cross-*processor* communication among clustered objects. Remote PPCs are used primarily for device interactions, for function shipping between the reps of a clustered object to coordinate their state where preferred over shared memory access (i.e., data shipping), and for operating on processor-local exception-level state (such as the per-processor PPC thread cache).

6.1 PPC implementation

The implementation of our PPC facility involves on-demand creation of server threads to handle calls and the caching of the threads for future calls. To maximize performance in a multiprocessor environment, state information about a PPC endpoint (a Tornado port), including a cache of ready threads, is maintained on a per-processor basis. This allows a full call and return to be completed without the need for any locks (beyond disabling interrupts as normally happens as part of the PPC trap) or accesses to shared data (in the common case that the port cache is not empty).

For each server that has previously been accessed on a processor, the processor maintains a list of worker threads to handle calls,which grows and shrinks according to the demand on the server on the processor. A PPC call involves just a trap, a couple of queue manipulations to dequeue the worker and enqueue the caller on the worker, and a return-from-trap to the server, with a similar sequence for a return PPC call. Parameter passing registers are left untouched by the call sequence and hence are implicitly passed between client and server.

On first use, as well as for the uncommon case of there being no workers available on the port, the call is redirected to a special *MetaPort*, whose sole purpose is to handle these special cases. The workers for this port have resources pre-reserved for them so that these redirected calls will always succeed.

Although the design is primarily targeted at maximizing concurrency, a common case call and return requires only 372 instructions, including 50 instructions for user-level register save and restore, 14 for stubs, 49 for the

clustered object security checking code, and 30 instructions for debugging. Stacks can optionally be mapped dynamically among all threads, which adds 62 instructions to the base latency, but allows the memory to be reused across servers, minimizing the cache footprint. This leaves 167 instructions for the core cost of a PPC call and return, which compares favorably to the corresponding cost of 158 instructions for two one-way calls for one of the fastest uniprocessor IPC systems running on the same (MIPS R4000 based) processor [20]. In addition, up to 8KB of data can be exchanged between client and server (in both directions) through remapping a region for an additional cost of only 82 instructions.

The final component of the PPC system, remote PPCs, are like regular PPCs with a pair of remote interrupts on the call and return to connect the two sides. One key difference, however, is that a full context switch is required on both sides for both the call and return. One natural optimization we have not yet applied would be to have the caller spin in the trap handler for a few microseconds before calling the scheduler in case the remote PPC call completes quickly, avoiding the overhead of the two context switches on the calling side. Still more expensive than we would like, the remote PPC call/return pair requires approximately 2200 instructions (including 1300 for two remote interrupt exchanges), plus the cost of four cache transfers (a pair for each of the remote PPC call and return exchanges).[17]

7 Experimental results

The results presented in this paper are based on both hardware tests on a locally developed 16 processor NUMAchine prototype [14, 31] and the SimOS simulator from Stanford [27]. The NUMAchine architecture consists of a set of *stations* (essentially small, bus-based multiprocessors) connected by a hierarchy of rings. It uses a novel selective broadcast mechanism to efficiently handle invalidations, as well as broadcast data. The test machine has 4 stations, each with 4 processors and a memory module (for a total of 16 processors), connected by a single ring. The final machine will have 48 processors with a two-level hierarchy of rings. The processors are 150MHz MIPS R4400 processors with 16K I/D L1 cache and 1MB L2 cache, all direct mapped. The bus and rings are 64 bits wide and clocked at 40MHz,[18] giving a bandwidth of 320MB/s for each link. The key latencies for the system are: 15 cycles for the secondary cache, 270 for local memory, and approximately 370 cycles for remote memory.

Although the simulator does not model precisely the same architecture as our hardware (in particular the interconnect and the coherence protocol implementation

[17] We expect to reduce this to about 600 instructions when optimizations already applied to the local case are applied to the remote call path.

[18] The final version will run at 50MHz.

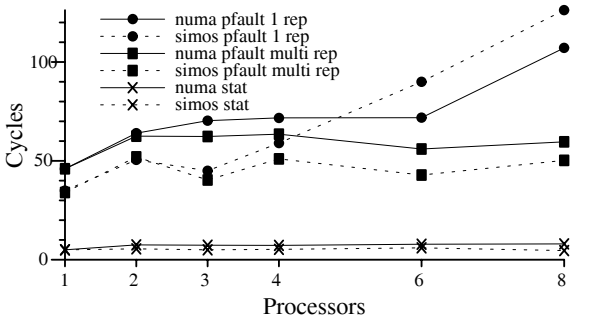

Figure 7: *Comparison of SimOS vs. NUMAchine for various benchmarks.*

are different), it does model a similar NUMA architecture based on the MIPS R4400 processor with component sizes, latencies, and bandwidths configured similar to those of NUMAchine. In addition, the simulator has been validated by previous researchers and by ourselves using both micro- and macro-benchmarks. For example, Figure 7 shows the results of a number of different tests run on SimOS and our hardware. (The tests are discussed in more detail below.) Although the exact cycle counts are not identical, the general trends match closely, allowing us to reasonably evaluate the effect of various tradeoffs in the Tornado architecture.

In this section we examine the performance of the individual components presented in this paper, and then look at some higher level microbenchmarks run under both Tornado and other current multiprocessor systems.

7.1 Component results

Figure 8(a) shows the results of a number of concurrent stress tests on the components described in this paper; namely, dynamic memory allocation (n threads malloc and free), clustered object miss handling (n threads invoke independent clustered objects not in the table), clustered object garbage collection (n threads trigger garbage collection), and protected procedure calling (n threads call a common clustered object in another address space). Results are collected over a large number of iterations and averaged separately for each thread. Figure 8(a) includes the average time in cycles across all threads, as well as range bars indicating the range of thread times.

These results show that memory allocation and miss handling perform quite well although there is a lot of variance across the threads for the garbage collection and PPC tests. As the number of processors increases, the range remains consistent, but the overall average slowly increases. If we compare the results from NUMAchine to those on SimOS, shown in Figure 8(b), we see the same sort of trend (except that it is worse under SimOS). However, running the same tests with SimOS set to simulate the caches with 4-way associativity, the results become almost perfectly uniform and flat (see Figure 8(c)). This indicates that the cause of the variability is local

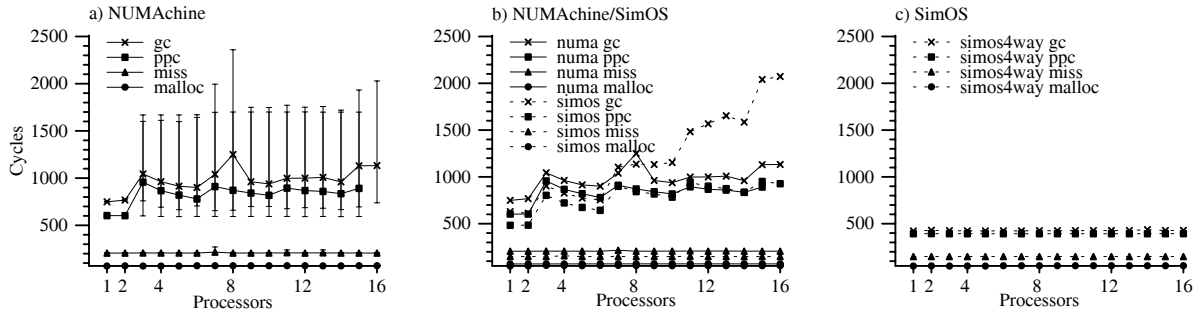

Figure 8: *Nanobenchmarks: Garbage collection (gc), Protected Procedure Calls (ppc), in-core page miss handling (miss), and dynamic memory allocation (malloc), run under Tornado on NUMAchine (numa) and under Tornado on SimOS (simos). The left figure (a) shows the average number of cycles required on NUMAchine for n threads with range bars indicating the range over all threads. The middle figure (b) shows the average cycles required on NUMAchine and SimOS. The right figure (c) shows the average cycles required on SimOS configured with 4-way set associative caches.*

cache conflicts—caused by multiple data structures occasionally mapping to the same cache block on some processors—and is not due to some unforeseen sharing.

7.2 Microbenchmarks

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

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

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

File Stat Each worker thread repeatedly fstats an independent file.

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

Figure 9(a) shows normalized results for the different tests on NUMAchine. Because all results are normalized against the uniprocessor tests, an ideal result would be a set of perfectly flat lines at 1. Overall, the results demonstrate good performance, since the slowdown is usually less than 50 percent. However, as with the component tests, there is high variability in the results, which accounts for the apparent randomness in the graphs.

Similar results are obtained under SimOS. Figure 9(b) shows the raw times in microseconds for the multithreaded tests run under NUMAchine and SimOS. As with the component tests, setting SimOS to simulate 4-way associative caches smooths out the results considerably (Figure 9(c)).[19]

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

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

Table 1: *Platforms on which micro-benchmarks where run.*

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

Table 2: *Base costs, in microseconds, for thread creation, page fault handling, and file stating, with a single processor.*

To see how existing systems perform, we ran the same tests on a number of systems available to us (see Table 1, Table 2, and Figure 10). The results demonstrate a number of things. First, many of the systems do quite well on the multiprogrammed tests, reflecting the effort that has gone into improving multi-user throughput over the last 10–15 years. However, the results are somewhat mixed for the multithreaded tests. In particular, although SGI does extremely well on the multiprogrammed

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

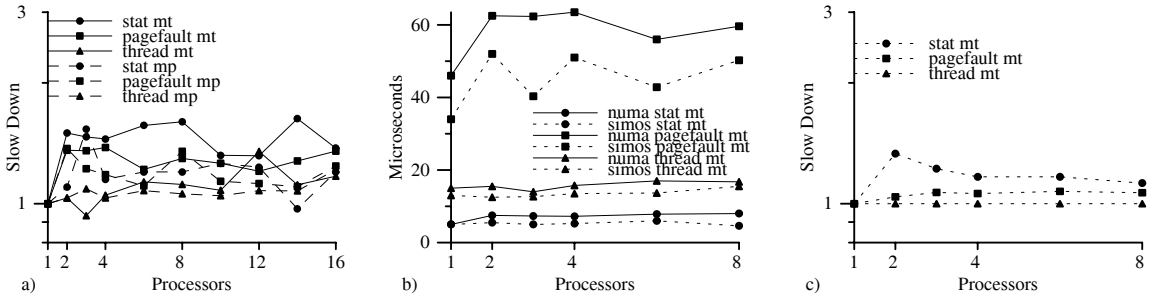

Figure 9: *Microbenchmarks: Cost of thread creation/destruction (thread), in-core page fault handling (pagefault), and file stat (stat) with n worker threads running either in one process (mt) or in n processes with one thread per process (mp). The left figure (a) shows slowdown relative to the uniprocessor case. The middle figure (b) shows the raw times in microseconds for the multithreaded tests on NUMAchine and SimOS, and the right figure (c) shows the slowdown of the multithreaded tests run on SimOS configured with 4-way set associative caches.*

tests, it does quite poorly on the multithreaded tests. This is particularly interesting when compared to results on an older bus-based, 6-processor SGI Challenge running IRIX 6.2 (not shown), where the multiprogrammed results are slightly worse and the multithreaded results are quite a bit better. Overall, Sun performs quite well with good multithreaded results and respectable multiprogrammed results; however, we only had a four processor system, so it is hard to extrapolate the results to larger systems.

One possible reason for poor performance is load balancing at a very fine granularity. For example, we suspect that the poor performance of AIX in the multiprogrammed thread creation experiment is due to a shared dispatch queue resulting in frequent thread migration. While load balancing is important, for most workloads systems like IRIX deal with it at a large granularity that does not require a shared dispatch queue.

7.3 Summary of results

While Tornado is still very much an active research project, the performance results obtained so far demonstrate the strengths of our basic design. The cycle counts provided throughout this paper for the costs of various operations demonstrates base performance competitive with commercial systems, which is important since scalability is of limited value if the base overhead is too large. In Section 7.1 we saw that the infrastructure of our system is highly scalable, including the IPC, clustered object, and the memory allocation facilities, providing the foundation for scalable system services. In Section 7.2 we saw that, for simple microbenchmarks, our system exhibits much better scalability than commercial systems. While microbenchmark results are generally considered a poor metric for comparison, the nearly perfect scalability of Tornado compares favorably to the large (e.g., 100X on 16 processors) slowdown for the commercial systems on the multithreaded experiments. It seems unlikely that this kind of a slowdown in the performance of such fundamental operations as page faults, thread creation, and file state does not have a large impact on application performance.

The disparity between the performance of the multithreaded and multiprogrammed results for the commercial systems suggests that locality and locking overhead on the operating system structures that represent a process is a major source of slowdown. The process is only a single example of a shared object, and we expect that experimentation will demonstrate that commercial systems exhibit slowdown (for even multiprogrammed experiments) when resources such as files or memory regions are shared. In our future work we will investigate the performance of Tornado when other resources are shared, and study the performance of our system for real applications.

8 Related work

A number of papers have been published on performance issues in shared-memory multiprocessor operating systems, mostly in the context of resolving specific problems in a specific system [5, 6, 9, 22, 26, 28]. These systems were mostly uniprocessor or small-scale multiprocessor systems trying to scale up to larger systems. Other work on locality issues in operating system structure were mostly either done in the context of earlier non-cache-coherent NUMA systems [8], or, as in the case of Plan 9, were not published [25]. Two projects that were aimed explicitly at large-scale multiprocessors were Hive [7], and the precursor to Tornado, Hurricane [30]. Both independently chose a clustered approach by connecting multiple small-scale systems to form either, in the case of Hive, a more fault tolerant system, or, in the case of Hurricane, a more scalable system. However, both groups ran into complexity problems with this approach and both have moved on to other approaches: Disco [4] and Tornado, respectively.

Clustered objects. Concepts similar to clustered objects have appeared in a number of distributed systems,

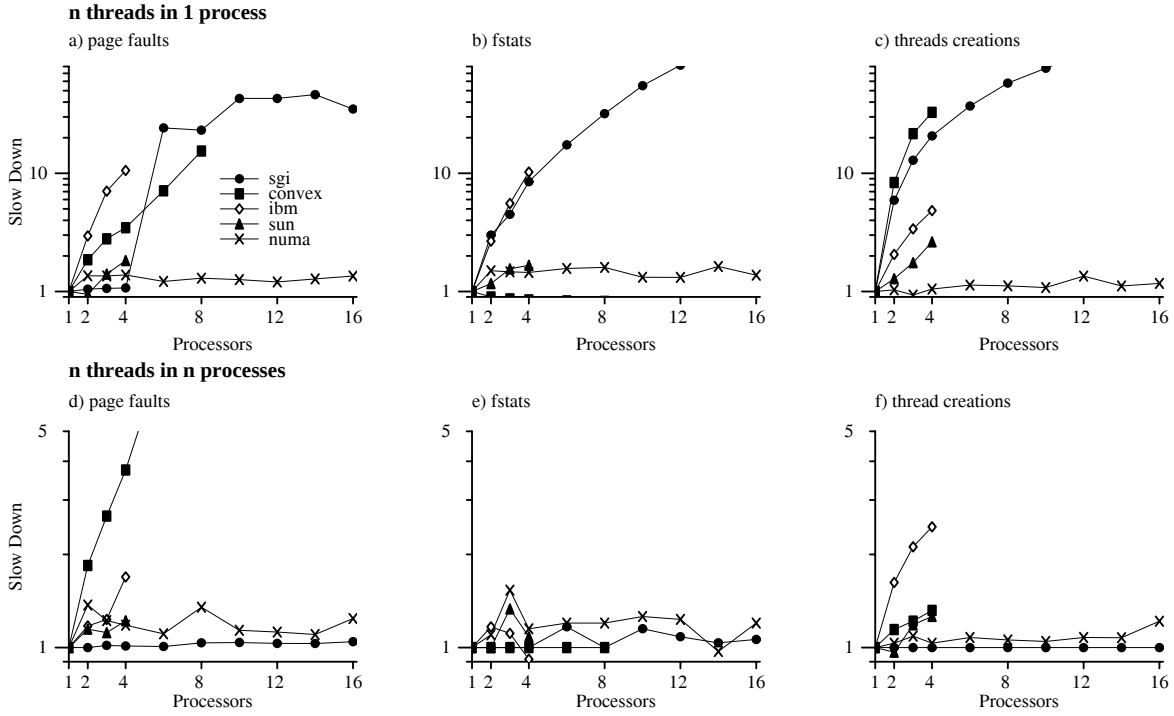

Figure 10: *Microbenchmarks across all tests and systems. The top row (a–c) depicts the multithreaded tests with n threads in one process. The bottom row (d–f) depicts the multiprogrammed tests with n processes, each with one thread. The leftmost set (a,d) depicts the slowdown for in-core page fault 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, and NUMAchine running Tornado.*

most notably in Globe [19] and SOS [21]. In all these cases (including Tornado's clustered objects) the goal is to hide the distributed nature of the objects from the users of the objects while improving performance over a more naive centralized approach. However, the issues faced in a tightly coupled shared-memory multiprocessor are very different from those of a distributed environment. For example, communication is cheaper, efficiency (time and space) is of greater concern, direct sharing is possible, and the failure modes are simpler. Hence, the Tornado clustered object system is geared more strongly towards maximizing performance and reducing complexity than the other systems.

Dynamic memory allocation. Our dynamic memory allocation design borrows heavily from McKenney and Slingwine's design [23], which is one of the few published works on multiprocessor memory allocation, in particular for kernel environments. A survey paper by Wilson et al [33] covers many of the other schemes, but does not address multiprocessor or caching issues. Grunwald et al examined cache performance of allocation schemes [16] and suggest a number of techniques they felt would be most effective in dealing with locality issues. Most of these techniques can be found in the McKenney and Slingwine memory allocator (with a few additions in our own adaptation).

Synchronization. The topic of locking and concurrency control in general has a long history, as does garbage collection [32]. The relationship between locking and garbage collection is evident in some of the issues surrounding memory management for lock-free approaches [18]. Our garbage collection scheme is in some sense a hack, but works reasonably well in our environment. Although it is somewhat similar to IBM's patent 4809168, their scheme appears to target uniprocessors only and is less general than ours. The benefits for our locking protocol are particularly evident in large, complex software systems, where there are many developers with varying skill and experience in dealing with concurrency control.

Protected procedure call. The majority of research on performance conscious inter-process communication (IPC) is for uniprocessor systems. Excellent results have been reported for these systems, to the point where it has been argued that the IPC overhead has become largely irrelevant [2].[20] Although many results have been reported

[20] We do not agree with these arguments.

over the years on a number of different platforms, the core cost for a call-return pair (with similar functionality) is usually between 100 and 200 instructions [11, 12, 17, 20]. However, the Tornado PPC facility goes beyond the standard techniques used to optimize IPC performance, by optimizing for the multiprocessor case by eliminating locks and shared data accesses, and by providing concurrency to the servers.

The key previous work done in multiprocessor IPC was by Bershad et al [3], where excellent results were obtained on the hardware of the time. However, it is interesting that the recent changes in technology lead to design tradeoffs far different from what they used to be. The Firefly multiprocessor [29] on which Bershad's IPC work was developed has a smaller ratio of processor to memory speed, has caches that are no faster than main memory (used to reduce bus traffic), and uses an updating cache consistency protocol. For these reasons, Bershad found that he could improve performance by idling server processes on idle processors (if they were available), and having the calling process migrate to that processor to execute the remote procedure. This approach would be prohibitive in today's systems with high cost cache misses and invalidations.

9 Concluding Remarks

Tornado was built on our experience with the Hurricane operating system [30]. Hurricane employed a course grained approach to scalability, where a single large scale SMMP was partitioned into clusters of a fixed number of processors. Each cluster ran a separate instance of a small scale SMMP operating system, cooperatively providing a single system image. This approach is now being used in one form or another by several commercial systems, for example in SGI's Cellular IRIX. However, despite many of the positive benefits of this approach, we found that: (i) the traditional within-cluster structures exhibit poor locality which severely impacts performance on modern multiprocessors, (ii) the rigid clustering results in increased complexity as well as high overhead or poor scalability for some applications, and (iii) the traditional structures as well as the clustering strategy make it difficult to support the specialized policy requirements of parallel applications.

Tornado does not have these problems. The object-oriented nature of Tornado and its clustered objects allow any available locality and independence to be exploited, allow the degree of clustering to be defined on a per-object basis, and make it easier to explore policy and implementation alternatives. Moreover, the fine-grained, in-object locking strategy of Tornado has much lower complexity, lower overhead, and better concurrency.

As the adage goes, "any problem in computer science can be solved by an extra level of indirection."[21] In Tornado, the clustered object translation table provides this level of indirection, which we have found useful for several purposes. For example, it includes clustered object access control information for implementing IPC security, helps track accesses to objects in support of the garbage collection system, and supplants the need for many other global tables by allowing clients to directly use references to server objects, rather than using an identifier that must be translated to the target object on each call.

The Tornado object-oriented strategy does not come without cost, however. Overheads include: (i) virtual function invocation, (ii) the indirection through the translation table, and (iii) the intrinsic cost of modularity, where optimizations possible by having one component of the system know about the details of another are not allowed. Our experiences to-date suggest that these costs are low compared to the performance advantages of locality, and will over time grow less significant with the increasing discrepancy between processor speed and memory latency. However, more experimentation is required.

Our primary goal in developing Tornado was to design a system that would achieve high performance on shared-memory multiprocessors. We believe that the performance numbers presented in this paper illustrate that we have been successful in achieving this goal. A result that is just as important that we did not originally target was ease of development. The object-oriented strategy coupled with clustered objects makes it easier to first get a simple correct implementation of a new service and then incrementally optimize its performance later. Also, the locking protocol has made it much easier for inexperienced programmers to develop code, both because fewer locks have to be acquired and because objects will not disappear even if locks on them are released.

Tornado currently runs on SimOS and on a 16 processor prototype of NUMAchine. It supports most of the facilities (e.g., shells, compilers, editors) and services (pipes, TCP/IP, NFS, file system) one expects. We are starting to explore scalability issues, work on policies for parallel applications, and study how to clusterize objects in a semi-automated fashion. A sister project, the Kitchawan operating system at IBM T.J.Watson Research Center, employs many of the ideas from Tornado, and is additionally exploring fault containment, availability, portability and some of the other issues required for an industrial strength operating system.

Acknowledgments

Rob Ho implemented most of the microbenchmarks. Ron Unrau and Tarek Abdelrahmen helped run the performance experiments. Many helped with the implementation of Tornado, in particular: Derek DeVries, Daniel Wilk, and Eric Parsons. Comments by Marc Auslander, Paul Lu, Karen Reid, Ron Unger, and David Wheeler.

[21] In a private communication, Roger Needham attributes this to

rau, and our shepherd Rob Pike helped improve the paper. Finally, this work was funded in part by IBM Corp. and the Natural Sciences and Engineering Research Council (NSERC).

References

[1] 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.

[2] B. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–212, 1992.

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

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

[5] M. Campbell et al. The parallelization of UNIX system V release 4.0. In *Proc. USENIX Technical Conference*, pages 307–324, 1991.

[6] J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proc. ACM SIGMETRICS Intl. Conf. on Measurement and Modelling of Computer Systems*, 1995.

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

[8] E. M. Jr. Chaves, P. C. Das, T. J. Leblanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, May 1993.

[9] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. Symp. on Operating Systems Design and Implementation (OSDI)*, pages 179–193, 1994.

[10] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

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

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

[13] B. Gamsa, O. Krieger, and M. Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proc. ICPP*, pages 208–211, 1994.

[14] A. Grbic et al. Design and implementation of the NUMAchine multiprocessor. In *Proceedings of the 35rd DAC*, pages 66–69, 1998.

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

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

[17] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proc. USENIX Summer Technical Conference*, 1993.

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

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

[20] T. Jaeger et al. Achieved IPC performance. In *6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

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

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

[23] P. E. McKenney and J. Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In *Proc. USENIX Technical Conference*, pages 295–305, 1993.

[24] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm. (De-)clustering objects for multiprocessor system software. In *Proc. Fourth Intl. Workshop on Object Orientation in Operating Systems (IWOOS95)*, pages 72–84, 1995.

[25] R. Pike. Personal communication.

[26] D. L. Presotto. Multiprocessor streams for Plan 9. In *Proc. Summer UKUUG Conf.*, pages 11–19, 1990.

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

[28] J. Talbot. Turning the AIX operating system into an MP-capable OS. In *Proc. USENIX Technical Conference*, 1995.

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

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

[31] Z. Vranesic et al. The NUMAchine multiprocessor. Technical Report CSRI-324, Computer Systems Research Institute, University of Toronto, 1995.

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

[33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Intl. Workshop on Memory Management*. Springer-Verlag, 1995.