Performance Issues for Multiprocessor Operating Systems

Benjamin Gamsa, Orran Krieger, Eric W. Parsons, Michael Stumm

Technical Report CSRI-339
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.

Performance Issues for Multiprocessor Operating Systems

Benjamin Gamsa Orran Krieger Eric W. Parsons Michael Stumm

Department of Computer Science
Department of Electrical and Computer Engineering
University of Toronto

1 Introduction

Designing an operating system for good performance is fundamentally more difficult for shared-memory multiprocessors than it is for uniprocessors. Multiprocessors require that a distinct set of issues be considered. For example, the overhead of cache consistency requires careful attention to the placement of data in order to reduce the number of cache misses. Similarly, in large systems the distribution of memory across the system also requires attention to the placement of data but this time to improve memory access locality. Dealing with these issues might involve such optimizations as data alignment, padding, regrouping, replication, and migration, or even the use of alternative data structures. Unfortunately, current compiler technology is not capable of performing many of these optimizations on code typical of operating systems. Hence, the systems programmer needs to be aware of the intricacies of multiprocessors and explicitly optimize the code for these systems.

This article first reviews how shared memory multiprocessor hardware affects the performance of system software and then presents a set of principles for organizing operating system data to maximize performance on these systems. The principles have been derived from a synthesis of our own experience developing (and performance tuning) two parallel operating systems and the experiences reported by other operating system developers. The principles are not only useful for adapting and performance tuning existing multiprocessor operating systems, but are also useful when designing a system from scratch. In the latter case, they lead to a system structure that is both novel and effective. As an example, we illustrate the use of these principles on selected components of a (complete and running) object-oriented operating system we have been implementing.

2 Multiprocessor Hardware Issues

In this section we describe how shared-memory multiprocessor hardware affects system software. In particular, we consider the effects of (i) true parallelism resulting from multiple processors, (ii) cache coherence and its protocols, and (iii) the high cache-miss latency of shared memory multiprocessors. The characteristics of the small-scale and large-scale multiprocessors we use for this analysis are described in Figure 1.

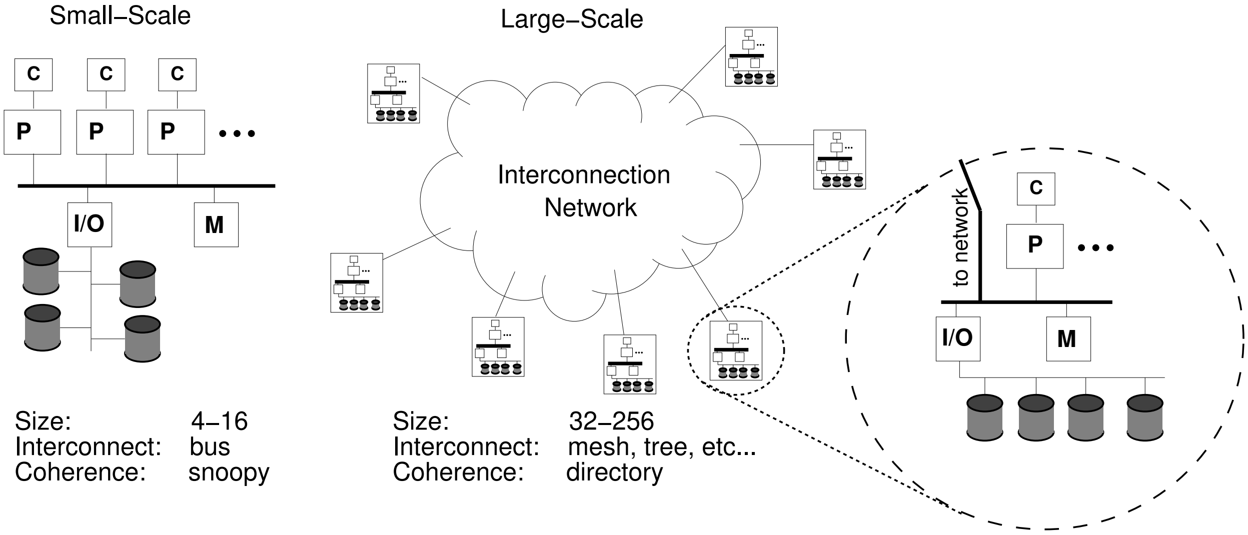

Figure 1: *This figure depicts the two basic classes of multiprocessors under consideration. The left figure shows a small-scale bus-based multiprocessor, which commonly range in size from 4-16 processors. The figure on the right shows the general architecture of a large-scale system, typically built from nodes containing a small number of processors, memory, and disks, and connected by some scalable interconnect (represented here by a cloud), which could be a mesh, tree, torus, etc. Important features common to both types of systems are the use of an invalidation-based coherence scheme (rather than the less common update-based scheme), and in-cache synchronization primitives (rather than network- or memory-based primitives).*

Since much of the material presented in the section is tutorial in nature, the well-informed reader should feel free to scan over it quickly.

2.1 Multiple processors

The most obvious difference between shared-memory multiprocessors (SMMPs) and uniprocessors is the number of processors. Although uniprocessor system software may already deal with concurrency issues, the true parallelism in multiprocessors introduces additional complications that can affect both the correctness and performance of uniprocessor synchronization strategies.

The strategies often employed for synchronization on uniprocessors, such as disabling interrupts in the kernel [26] or relying on the non-preemptability of a server thread [15], are not directly applicable on multiprocessors. Although they can be made to work by allowing only a single processor to be in the kernel at a time (or a single process to be executing in a server at a time), this serializes all requests and is not acceptable for performance reasons. For example, gcc spends over 20 percent of its time in the kernel under Ultrix [8], which would limit the useful size of a multiprocessor to 5 processors if all kernel requests were serialized. As a result, a fully preemptable (and fully parallelized) system software base is generally required [9, 29].

Fine-grained locks are generally needed to achieve a high level of concurrency and improved performance; however, the finer the granularity of locks, the larger the number of locks that must be acquired to complete an operation, resulting in higher overhead even if there is no contention for the locks. For example, in a previous system, we found that locking overhead in the fully uncontended case accounted for as much as 25 percent of the total page-fault handling time [32]. It is therefore necessary to carefully balance the desire for high concurrency through finer-grained locks, against the desire for lower overhead through coarser-grained locks.

In addition, there are complex tradeoffs to consider in the implementation of locks on a multiprocessor. Whereas the choice is straightforward for uniprocessors where the only sensible option is a simple blocking lock, multiprocessor locking must consider such issues as context-switch overhead, wasted spin-cycles, bus and network contention due to spinning, fairness, and preemption effects for both lock holders and spinners. Fortunately, this area has been well-studied [2, 3, 5, 13, 18, 32], and in most cases spin-then-block locks with exponential back-off for the spinning phase have proven to be highly effective [13].[1]

2.2 Cache coherence

Software on a shared-memory multiprocessor suffers not only from cold, conflict, and capacity cache misses (as on a uniprocessor), but also from *coherence* misses. Coherence misses are caused by read/write sharing and the cache-coherence protocol,[2] and are often the dominant source of cache misses. For example, in a study of IRIX on a 4-processor system, Torrellas found that misses due to sharing dominated all other types of misses, accounting for up to 50 percent of all data cache misses [30]. In addition to misses caused by direct sharing of data, writes by two processors to distinct variables that reside on the same cache line will cause the cache line to ping-pong between the two processors. This problem is known as false sharing and can contribute significantly to the cache miss rate. Chapin et al., in investigating the performance of IRIX ported to a 32 processor experimental system, found that many of the worst-case hot spots were caused by false sharing [7]. Although strategies for dealing with misses in uniprocessors by maximizing temporal and spatial locality and by reducing conflicts are well known, if not always easily applied, techniques for reducing true and false sharing misses in multiprocessors are less well understood. Semi-automatic methods for reducing false sharing, such as structure padding and data regrouping, have proven somewhat effective, but only for parallel scientific applications [16, 31].

The effects of synchronization variables can have a major impact on cache performance, since they can have a high degree of read/write sharing and often induce large amounts of false sharing. For example, Rosenblum noted in a study of an 8 processor system that 18 percent of all coherence misses were caused by the false sharing of a single cache line containing a highly shared lock [27]. Even without false sharing, the high cost of cache misses can increase the cost of a lock by an order of magnitude over the fully cached case.[3]

2.3 Cache miss latency

In addition to the greater frequency of cache misses due to sharing, SMMP programmers are faced with the problem that cache misses cost more in multiprocessors regardless of their cause. The latency increase of cache misses on multiprocessors stems from a number of sources. First, more complicated controllers and bus protocols are required to support multiple processors and cache coherence, which can increase the miss latency by a factor of two or more.[4] Second, the probability of contention at the memory and in the interconnection network (bus, mesh, or other) is higher in a multiprocessor. Extreme examples can be found

[1] Distributed queue-based spin locks are another well-studied option, whose prime value is for the less common cases that can benefit from continual spinning (even under high contention) and which also require low-latency and a high degree of fairness [20, 23].

[2] To be more precise, invalidation-based cache coherence protocols require that the cache obtain exclusive access to a line the first time it is written to, even if no sharing is taking place. These misses are sometimes referred to as *upgrade* misses or *initial-write* misses. Hence there is an extra cost in multiprocessors both for sharing misses and upgrade misses.

[3] Some results suggest that there is often sufficient locality that the locks can generally be assumed to reside in the cache [30]. However, later work (by the same author) suggests that coherence traffic due to locking is a significant problem [35]. The effects of false sharing may well explain this apparent contradiction.

[4] Even the Digital 8400 multiprocessor, which is expressly optimized for read-miss latency, has a 50 percent higher latency than Digital's earlier, lower performance, uniprocessors [10].

in early work on Mach on the RP3 where it took 2.5 hours to boot the system due to memory contention [6], and in the work porting Solaris to the Cray SuperServer where misses in the idle process slowed non-idle processors by 33 percent [21]. Third, the directory-based coherence schemes of larger systems must often send multiple messages to retrieve up-to-date copies of data or invalidate multiple sharers of a cache-line, further increasing both latency and network load. This effect is clearly illustrated in the operating system investigation by Chapin et al., which found that local data miss costs were twice the local instruction cache miss costs, due to the need to send multiple remote messages [7].

In the case of large systems, the physical distribution of memory has a considerable effect on performance. Such systems are generally referred to as NUMA systems, or Non-Uniform Memory Access time systems, since the time to access memory varies with the distance between the processor and the memory module. In these systems, physical locality plays an important role in addition to temporal and spatial locality. An example can be seen in the work of Unrau et al., where, due to the lack of physical locality in the data structures used, the uncontended cost of a page fault increased by 25 percent when the system was scaled from 1 to 16 processors [33].

As a result of the high cost of cache misses, newer systems are being designed to support non-blocking caches and write buffers in order to hide load and store latencies. Many of these systems allow loads and stores to proceed while previous loads and stores are still outstanding. This results in a system in which the order of loads and stores may not appear the same to each processor; these systems are sometimes referred to as being weakly consistent [12].

The implication of weakly consistent systems is that regular memory accesses cannot be used for synchronization purposes, since memory access can appear out of order. For example, if one processor fills a buffer and marks it filled, other processors might see the buffer first being marked filled and then actually filled with data. This could cause some processors to access stale data when they see the buffer marked filled. As a result of this behavior, software must generally switch from using simple flags to using full locking, with significantly more expensive special memory synchronization instructions. This difference is illustrated by the cost of a lock/unlock pair in AIX on the PowerPC, which is 100 times more expensive that a cached store [29].[5]

Another result of the high cache-miss latency is the movement towards larger cache lines of 128 or even 256 bytes in length. These large cache lines are an attempt to substitute bandwidth (which is relatively easy to design into a system) for latency (which is much harder to design in). Unfortunately, large cache lines tend to degrade performance in SMMPs due to increased false sharing.[6] The example cited earlier of a single falsely-shared cache line being responsible for 18 percent of all coherence misses was due in part to the fact that the hardware had 128 byte cache lines, causing a critical lock to be shared with 22 other randomly placed variables [27].

2.4 Summary

The memory access times in a multiprocessor span several orders of magnitude due to architectural and contention effects (see Table 1). To mitigate these effects, system software must be structured to reduce cache misses and increase physical locality. In the next section we present a set of structuring techniques and the circumstances under which they should be applied.

[5] In principle, locks would not be required for single flag variables if it were possible to insert (or have the compiler insert) the required memory barrier instructions at the right places, but such support is not generally available at this time [1].

[6] To further complicate matters, with such a wide variety of cache line sizes (sometimes even within the same product line), ranging from 16 to 256 bytes, it is becoming increasingly difficult to optimize for a fixed or "average" cache line size.

Access Type	Latency
Primary cache	1
Secondary cache	10
Local memory	100
Remote memory	200–500
Hot spot	1000–10000

Table 1: *Latencies for different types of accesses in a typical large-scale shared-memory multiprocessor, measured in processor cycles.*

3 Design principles

The issues described in the previous section clearly have a major impact on the performance of multiprocessor system software. In order to achieve good performance, system software must be designed to take the hardware characteristics into account. There is no magic solution for dealing with them; the design of each system data structure must take into account the expected access pattern, degree of sharing, and synchronization requirements. Nevertheless, we have found a number of principles and design strategies that have repeatedly been useful. These include principles previously proposed by us and others [7, 32, 33, 35], refinements of previously proposed principles [14] to address the specific needs of system software, and a number of new principles.

3.1 Structuring data for caches

When frequently accessed data is shared, it is important to consider how the data is mapped to hardware cache lines and how the hardware keeps the cached copies of the data consistent. Principles for structuring data for caches include:

S1: *Segregate read-mostly data from frequently modified data.* Read-mostly data should not reside in the same cache line as frequently modified data in order to avoid false sharing. Segregation can often be achieved by properly padding, aligning, or regrouping the data. Consider a linked list whose structure is static but whose elements are frequently modified. To avoid having the modifications of the elements affect the performance of list traversals, the search keys and link pointers should be segregated from the other data of the list elements.[7]

S2: *Segregate independently accessed read/write data from each other.* This principle prevents false sharing of read/write data, by ensuring that data that is accessed independently by multiple processors ends up in different cache lines.

S3: *Privatize write-mostly data.* Where practical, generate a private copy of the data for each processor so that modifications are always made to the private copy and global state is determined by combining the state of all copies. This principle avoids coherence overhead in the common case, since processors update only their private copy of the data. For example, it is often necessary to maintain a reference count on an object to ensure that the object is not deleted while it is being accessed. Such a reference count can be decomposed into multiple reference counts, each updated by a different processor and, applying **S2**, forced into separate cache lines.

[7] This is in marked contrast to uniprocessors, where it is better to collocate the linked list state and the list elements so that when an element is reached some of its data will have been loaded into the cache.

S4: *Use strictly per-processor data wherever possible.* If data is accessed mostly by a single processor, it is often a good idea to restrict access to only that processor, forcing other processors to pay the extra cost of inter-processor communication on their infrequent accesses. In addition to the caching benefits (as in **S3**), strictly per-processor structures allow the use of uniprocessor solutions to synchronize access to the data. For example, for low-level structures, disabling interrupts is sufficient to ensure atomic access. Alternatively, since data is only accessed by a single processor, the software can rely on the ordering of writes for synchronization, something not otherwise possible on weakly consistent multiprocessors (see Section 2.3).

3.2 Locking data

In modern processors, acquiring a lock involves modifying the cache line containing the lock variable. Hence, in structuring data for good cache performance, it is important to consider how accesses to the lock interact with accesses to the data being locked.

L1: *Use per-processor reader/writer locks for read-mostly data.* A lock for read-mostly data should be implemented using a separate lock for each processor. To obtain a read-lock, a processor need only acquire its own lock, while to obtain a write-lock it must acquire all locks. This strategy allows the processor to acquire a read-lock and access the shared data with no coherence overhead in the common case of read access. (This principle can be viewed as a special case of principle **S3**.)

L2: *Segregate contended locks from their associated data if the data is frequently modified.* If there is a high probability of multiple processes attempting to modify data at the same time, then it is important to segregate the lock from the data so that the processors trying to access the lock will not interfere with the processor that has acquired the lock.[8]

L3: *Collocate uncontended locks with their data if the data is frequently modified.* When a lock is brought into the cache for locking, some of its associated data is then brought along with it, and hence subsequent cache misses are avoided.

3.3 Localizing data accesses

For large-scale systems, the system programmer must be concerned with physical locality in order to reduce the latency of cache misses, to decrease the amount of network traffic, and to balance the load on the different memory modules in the system. Physical locality can be especially important for operating systems since they typically exhibit poor cache hit rates [8].

D1: *Replicate read-mostly data.* Read-mostly data should be replicated to multiple memory modules so that processors' requests can be handled by nearby replicas. Typically, replication should occur on demand so that the overhead of replicating data is only incurred if necessary.

D2: *Partition and migrate read/write data.* Data should be partitioned into constituent components according to how the data will be accessed, allowing the components to be stored in different memory modules. Each component should be migrated on use if it is primarily accessed by one processor at a time. Alternatively, if most of the requests to the data are from a particular client, then the data should be migrated with that client.

[8] Special hardware support for locks has been proposed that results in no cache-coherence traffic on an unsuccessful attempt to acquire a lock, making principle L2 unnecessary [17].

D3: *Privatize write-mostly data.* Privatizing write-mostly data, as described in principle **S3**, can be used not only to avoid coherence overhead, but also to distribute data across the system to localize memory accesses.

Although the following principles are not strictly for structuring data for locality, we have found them equally important in achieving efficient localization of operating system data.

D4: *Use approximate local information rather than exact global information.* For certain operating system policies, it is possible to sacrifice some degree of accuracy in exchange for performance by using local approximate information to make reasonable decisions rather than exact global information. For example, having per-processor run-queues reduces short-term fairness globally, but minimizes the cost to the dispatcher when a process is being scheduled to run.

D5: *Avoid barrier-based synchronization for global state changes.* When data is replicated or partitioned, it is necessary to synchronize when making global changes to the replicas and when determining a globally consistent value for the partitioned data. In this case, the system should avoid using barrier-based synchronization because it wastes processor cycles while waiting for other processors to reach the barrier, and results in a high overhead in interrupting (and restarting) the tasks running on the processors [35]. There are a variety of alternative *asynchronous* schemes (e.g., as used for lazy TLB shoot-down [25]) which can, for example, allow multiple requests to be combined together to reduce the amount of inter-processor communication.

In applying the **S***, **L***, and **D*** principles, the programmer must be aware that their over-zealous application may actually reduce performance. For example, while a naive application of the **S*** principles may result in a system with reduced coherence overhead, it may also result in an increased total number of cache misses due to the fragmentation of data structures. Hence, significant expertise is required to apply the design principles in a balanced fashion.

4 Application of the design principles

To illustrate how the design principles influence system software design, we describe selected parts of Tornado [24], an object-oriented operating system for the NUMAchine [34] large-scale multiprocessor. First, we describe our basic model of objects, one that facilitates the application of the design principles across our system. Then, we describe four components of our infrastructure to support this model, each of which uses the design principles in its internal implementation.

4.1 Structuring objects

The traditional approach for implementing objects in uniprocessor systems does not extend well to multiprocessor systems. A uniprocessor object is typically implemented as a single structure, organized to maximize spatial and temporal locality, and possibly protected by a single lock to serialize concurrent accesses. In a multiprocessor, such an object may experience significant coherence overhead due to the active sharing of frequently-modified cache lines and may not offer sufficient concurrency. Moreover, the object will be located in a single memory module, which in a large-scale system will cause remote processors to experience high access latencies.

One way to achieve good multiprocessor performance is to partition object data into distinct components so as to reduce cache coherence overhead (**S***), to allow each component to be locked independently (**L1**), and to allow each component to be placed close to processors that are accessing it (**D***). However, this requires a mechanism that allows a processor to efficiently locate the component of the object that it should access. Although an object could provide a redirection table for this purpose, a naive implementation would require all accesses to pass through a centralized table, thereby limiting the benefits of distributing the data of the object. The approach we have taken is based on a new model of objects, called *clustered objects*, that allows objects to be easily partitioned while preserving the fundamental object abstraction [24].

4.2 Clustered object model

A clustered object is an object implemented by a set of *representative objects*, which all support a common interface (i.e., that of the object); these representatives are typically distributed across a system, serving as different (local) points of access. In this model, part of the clustered object data may be replicated across its representatives, while other parts may be maintained exclusively by one representative, possibly being migrated from representative to representative as it is accessed. For each object, a processor is associated with a particular representative, to which it directs method invocations. The representatives together are responsible for giving their clients the view of a single, consistent object, hiding the fact that the object is partitioned and distributed. (Note that since the clustered object model serves only as a framework for partitioning objects, the programmer must still write code to associate representatives and processors, distribute representatives to memory modules, and segregate and/or replicate data across the representatives.)

As an example of a clustered object, consider a kernel object that implements a region of memory that is mapped into an application's address space. Assume that the application is operating on a matrix in a partitioned manner, with only a few boundary pages shared between processes (see upper portion of Figure 2). The application's partitioned access pattern allows design principles **S2** and **D2** to be applied in partitioning the data of the region object. In this example, a region representative is created for each of the four processors, which records the local page mappings and manages the physical cache for the matrix (see lower portion of Figure 2). When a process experiences a page fault or prefetches a page, the processor first locates its region representative. This representative then, in the common case, allocates a page in local memory and directs the pager to fetch the corresponding page. If the page is one of those that is shared, then the representative forwards the request to the representative responsible for managing that page. In this implementation, a separate set of internal data structures (linked lists, hash tables, etc.) exists for each representative (**S4**), while each page descriptor is managed by a single representative (**D2**). This organization allows most requests to be handled solely by a processor's dedicated representative, resulting in no lock contention, no remote memory accesses, and improved cache performance.

4.3 Clustered object support

Since all components of our system rely heavily on clustered objects in applying the design principles, it is vital that the services used to support clustered objects be implemented efficiently. Our basic approach in designing these services is to have the key data structures required to handle common requests maintained on a per-processor basis, applying design principles **S4**, **D1**, **D2**, and **D3**. For less common operations, we typically rely on the other principles to minimize overheads.

In this section, we describe the implementations for invoking clustered object methods, for destroying clustered objects, for accessing objects across address spaces, and for dynamically allocating memory, focussing on those aspects of our implementation that illustrate the design principles.

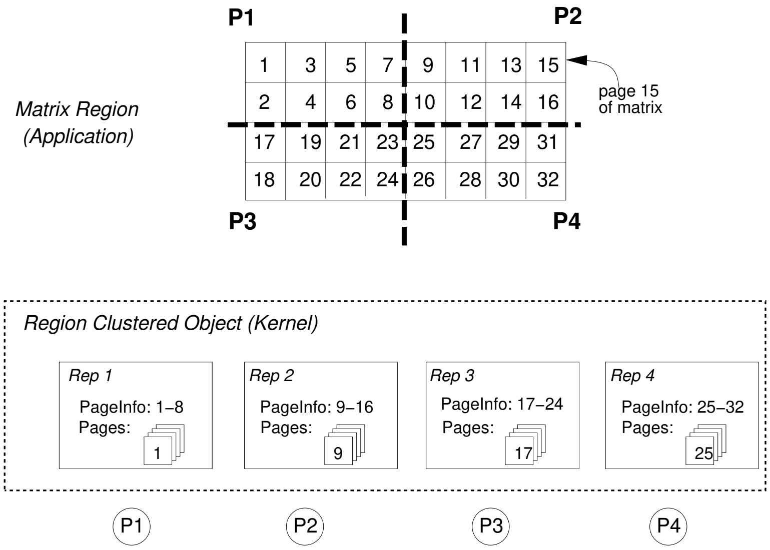

Figure 2: *This figure illustrates an application accessing a matrix in a partitioned fashion, and the corresponding implementation of the region in the kernel. Each processor possesses its own region representative (Rep 1 through Rep 4), stored in processor-local memory, to which page fault and prefetch requests are directed. These representatives keep track of the physical pages allocated to the region as well as the virtual-to-physical mappings. Since each representative manages different portions of the region, a representative can be locked without delaying the progress of other processors.*

4.3.1 Clustered-object invocation

A clustered object is identified by a unique object identifier (OID), which is translated to a reference to the appropriate representative on each invocation. To perform this translation efficiently, we allocate on each processor an object translation table which records references to the representatives to be used by that processor (**D2**) (see Figure 3). Since all translation tables are aliased to the same virtual address on every processor, invoking a method on a clustered object only requires the added step of indexing into the object translation table using the OID. A translation table entry is initialized the first time an object is accessed by a processor, which may in some cases also trigger the allocation of a new (nearby) representative.[9]

Given this scheme, a clustered-object invocation does not commonly entail remote memory references. In addition, invocations can typically be performed without explicit synchronization because the per-processor data structures are only modified by remote processors by interrupting the target processor (applying principle **S4**). This allows a clustered object method invocation to require only one extra instruction over the non-clustered-object case.[10]

4.3.2 Clustered object destruction

The technique for destroying clustered objects is somewhat involved. To make it possible to safely invoke a clustered object without worrying about the object's destruction, we constrain destruction to occur only when no processes are actively accessing the object. Additionally, to avoid the use of costly barriers, we

[9] More detailed aspects of the implementation are described elsewhere [24].
[10] Our system is based on C++, and hence the cost to invoke a non-clustered-object is the cost of a C++ virtual function call.

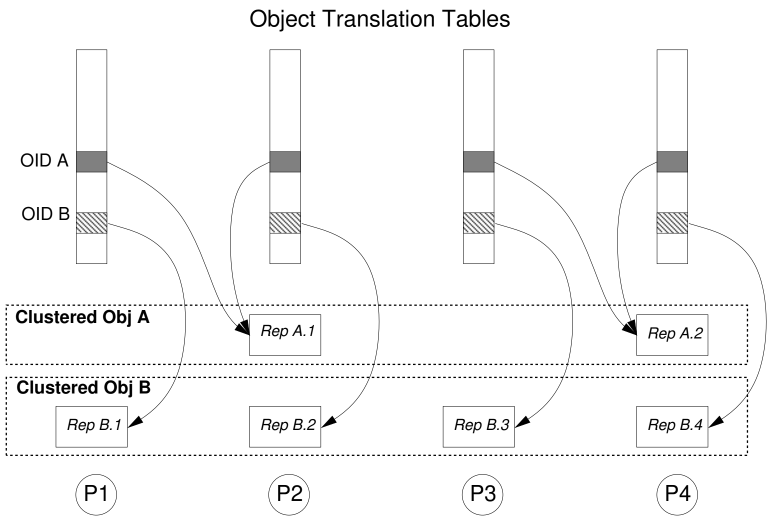

Figure 3: *The clustered object framework allows a clustered object to associate a different representative with each processor (and share representatives among processors if desired). This relationship is stored in a per-processor object translation table. In this way, an object identifier (OID) can be transparently used anywhere in the system; each access will be directed to the appropriate object representative. For scalability, the table entries are filled on demand, so although all the entries are shown as if they were filled, they would only be filled if all processors were actually accessing the two objects shown.*

use an asynchronous technique for object destruction (**D5**). This ensures that processors are never blocked (avoiding wasted cycles spinning), and allows destruction-requests to be batched (reducing inter-processor communication).

A reference count scheme is used to determine when an object's memory can be reclaimed. Rather than maintaining a separate reference count for each object, a single per-processor reference count keeps track of the number of threads accessing any clustered object on the processor (**L1, S3, D3**). When a thread first enters an address space to invoke a method on a clustered object, it increments the counter for the processor; subsequent object invocations within the address space require no further increments.

Destroying an object is performed in two phases. In the first phase the object invalidates all references to itself held by other objects and notifies the appropriate set of processors that it is in the process of being destroyed. When a processor's reference count reaches zero, it invalidates its translation table entry; until this point, it is possible that a thread running on that processor will have made a temporary copy of the reference (say, in a register), and hence be capable of making new method invocations to the object. The second phase starts once all translation table entries have been invalidated; it consists of lazily reclaiming the memory used by the representatives.[11]

4.3.3 Cross-address-space invocation

In order to preserve the performance advantages of using clustered objects across address spaces, a cross-address-space communication service must be available that exhibits a similarly high degree of locality

[11] This object destruction algorithm is similar in nature to lazy TLB shoot-down algorithms that re-use physical pages only once all processors are known to have discarded their associated TLB entry [25].

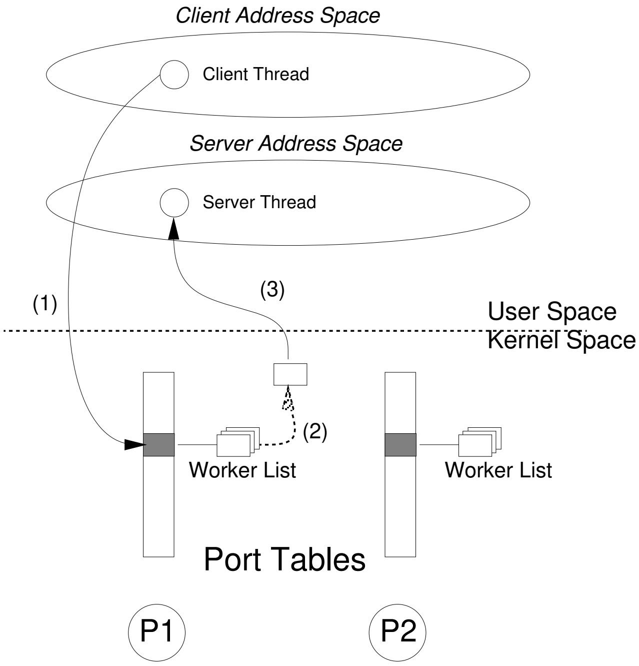

Figure 4: *The kernel maintains a per-processor table for the ports corresponding to servers in the system. When a client makes a cross-address space method invocation to such a port, it first traps into the kernel (1). The kernel then removes a pre-initialized worker (stored in processor-local memory) from the worker list for the port (2). It then continues execution in the server's address space (3) at the object translation entry point. When the server thread terminates, the worker structure in the kernel is returned to the port's worker list.*

and concurrency [11]. We use per-processor data structures for common-case operations, resulting in the fastest multiprocessor cross-address-space method invocation we know of, matching that of any uniprocessor implementation [19].

To accept cross-address space requests, a server must first allocate a port from the kernel to be used by clients in making requests. Per-processor tables are used to maintain port information, which includes information about the allocating server (**D1**) as well as a private pool of pre-initialized worker process descriptors (PDs) that reside in local memory (**D2**), as shown in Figure 4.[12] Handling a cross-address-space request consists of removing the first PD from the processor's private worker queue, and (possibly) re-mapping a page of memory to pass parameters that cannot fit in registers. Hence, this implementation of cross-address-space method invocation requires no remote data accesses and no locking (given that the kernel performs this at interrupt level), resulting in a null method call requiring only $\sim$300 instructions.

4.3.4 Memory allocation

Our system memory allocator is adapted from an earlier allocator designed for a small-scale multiprocessor [22]. This service is important for two reasons. First, object-oriented programs tend to use dynamic memory allocation generously [4]. Second, being able to allocate memory local to a processor is vital for

[12] The importance of having local worker structures has been observed in numerous other contexts, including network protocol processing [28] and process dispatching [30].

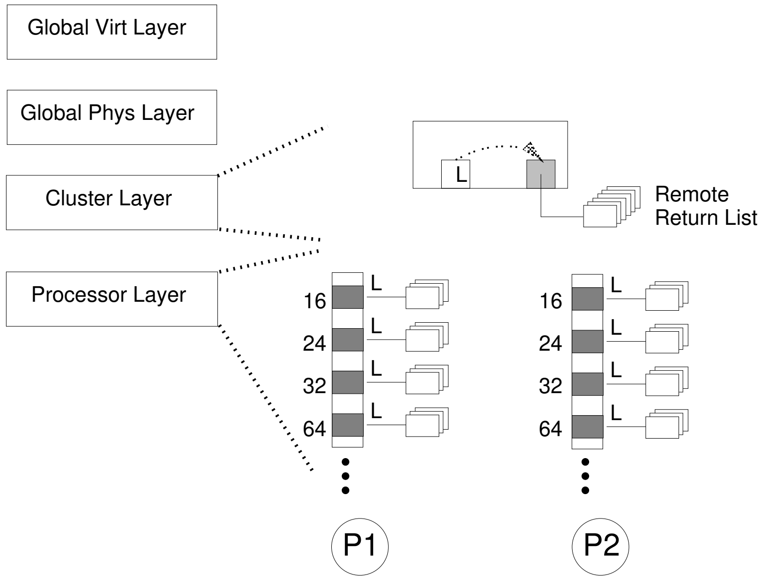

Figure 5: *The kernel memory allocator consists of four layers: the processor layer manages blocks of memory for each processor; the cluster layer balances the allocation of memory blocks among a cluster of processors sharing a common memory module; the global physical layer manages the physical pages available in the system; finally, the global virtual layer manages the allocation of virtual space for memory allocation.*

Because the bottom two layers are accessed most frequently, we concentrate on these. Each processor maintains a list of free blocks for memory of various sizes. These are used to allocate and free memory blocks of a particular size on the given processor. Since contention is expected to be low, each list is protected by a lock embedded in the head pointer for the list.

If a processor frees memory that belongs to a different processor, however, the memory is returned to a return list at the cluster level, which then redistributes the blocks in bulk. Since there might be contention in this case, the lock is maintained separately from the pointer to the list.

implementing clustered objects.

Our focus is on the bottom two layers of the allocator, as these are accessed most frequently (see Figure 5). To manage sharing and maximize locality, per-processor queues of memory blocks for each block size are used as free lists in the lowest, per-processor layer (**D2**). Since contention is rare (occurring only when a processor is preempted while manipulating a queue), design principle **L3** applies and we place a lock with the pointer to the head of each queue.

Free lists may not be modified by remote processors in order to reduce the number of cache misses and lock contention. In the rare case that a remote processor wishes to free data, it must use a separate remote-return-list, maintained at the cluster layer, thus applying design principle **S2**. In contrast to the processor-layer lists, the remote-return-list might experience contention, so applying design principle **L2**, we keep the lock in a separate cache line from the list pointer. Even though this requires accessing two cache lines to free a block, the cost is minor compared to having accessed the data remotely in the first place and is warranted by the potential reduction in cache coherence traffic.

5 Concluding remarks

We have described the effects of multiprocessor hardware on the performance of system software, provided a number of design principles that are useful in optimizing multiprocessor system software, and shown how these design principles can be applied to real software. Our focus throughout has been on low-level issues related to the organization of system data structures. In doing so we have not addressed many other important multiprocessor issues. For example, we did not discussed tradeoffs between different synchronization algorithms, or how code should be distributed across the system for locality.

Multiprocessor operating system programmers also face high-level challenges, such as, developing techniques to manage the complex resources of large-scale systems and handling the resource requirements of parallel programs. While it is crucial to address these high-level challenges, the benefits can only be fully realized if the basic operating system data is properly structured.

The work described is part of a large project involving the design and implementation of the NUMAchine multiprocessor, the Tornado operating system, and a variety of tools, compilers, and applications. For high application-level performance it is necessary that all levels involved perform well including the hardware, the operating system, and the application. In this paper we have described some of the techniques we have found to be important in achieving high performance at the operating system level.

References

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report ECE Technical Report 9512, Rice University, September 1995. Also as Digital Western Research Laboratory Research Report 95/7.

[2] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] David L. Black, Avadis Tevanian Jr. , David B. Golub, and Michael W. Young. Locking and reference counting in the mach kernel. In *Proc. 1991 ICPP*, volume II, Software, pages II–167–II–173, August 1991.

[4] Brad Calder, Dirk Grunwald, and Ben Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[5] M. Campbell, R. Holt, and J. Slice. Lock granularity tuning mechanisms in SVR4/MP. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II.)*, pages 221–228, March 1991.

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

[7] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conferentce on Measurement and Modelling of Computer Systems*, May 1995.

[8] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th ACM SOSP*, pages 120–133, 1993.

[9] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, D. Stein, M. Smith, A. Shivalingiah, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing: Multithreading the System V Release 4 kernel. In *USENIX Conference Proceedings*, pages 11–18, Summer 1992.

[10] David M. Fenwick, Denis J. Foley, William B. Gist, Stephen R. VanDoren, and Daniel Wissell. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

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

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, page 15, June 1990.

[13] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of the performance of parallel application. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, 1991.

[14] Mark D. Hill and James R. Larus. Cache considerations for mutliprocessor programmers. *CACM*, 33(8):97–102, August 1990.

[15] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[16] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 179–188, July 1995.

[17] Alain Kagi, Nagi Aboulenein, Douglas C. Burger, and James R. Goodman. An analysis of the interactions of overhead-redudincg techniques for shared-memory multiprocessors. In *Proc. International Conference on Supercomputing*, pages 11–20, July 1995.

[18] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 41–55, October 1991.

[19] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 175–187, 1993.

[20] Peter Magnussen, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *8th IPPS*, pages 26–29, 1994.

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

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

[23] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Comp. Sys.*, 9(1):21–65, Feb. 1991.

[24] Eric Parsons, Ben Gamsa, Orran Krieger, and Michael Stumm. (De-)Clustering objects for multiprocessor system software. In *Fourth International Workshop on Object Orientation in Operating Systems 95 (IWOOO'95)*, pages 72–81, 1995.

[25] J. Kent Peacock, Sunil Saxena, Dean Thomas, Fred Yang, and Wilfred Yu. Experiences from multithreading System V Release 4. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pages 77–92, March 1992.

[26] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *CACM*, 17(7):365–375, July 1974.

[27] Mendel Rosenblum, Edouard Bugnion, Emmett Witchel Stephen A. Herrod, and Anoop Gupta. The impact of architectural trends on operating system performance. In *To Appear in The 15th ACM Symposium on Operating Systems Principles*, December 1995.

[28] James D. Salehi, James F. Kurose, and Don Towsley. The performance impact of scheduling for cache affinity in parallel network processing. In *Fourth IEEE International Symposium on High Performance Distributed Computing*, August 1995.

[29] Jacques Talbot. Turning the AIX operating system into an MP-capable OS. In *USENIX 1995 Technical Conference Proceedings*, January 1995.

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

[31] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Transactions on Computers*, 43(6), June 1994.

[32] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Operating System Design and Implementation*, pages 139–152, November 1994.

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

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

[35] Chun Xia and Josep Torrellas. Improving the data cache performance of multiprocessor operating systems. In *To appear in HPCA-2*, 1996.