RC22863 (W0307-200) July 30, 2003
Computer Science

IBM Research Report

Enabling Scalable Performance for General Purpose Workloads on Shared Memory Multiprocessors

Jonathan Appavoo[1], Marc Auslander[1], Dilma Da Silva[1], Orran Krieger[1], Michal Ostrowski[1], Bryan Rosenburg[1], Robert W. Wisniewsk[1], Jimi Xenidis[1], Michael Stumm[2], Ben Gamsa[2], Reza Azimi[3] Raymond Fingas[3], Adrian Tam[3] David Tam[3]

[1]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[2] University of Toronto,
Toronto, Ontario, Canada, M5S 1A1

[3] SOMA Networks, Inc.

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g. , payment of royalties). Copies may be requested from IBM T. J. Watson Research Center ,
P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Enabling Scalable Performance for General Purpose Workloads on Shared Memory Multiprocessors.

Jonathan Appavoo † Marc Auslander † Dilma Da Silva † Orran Krieger †
Michal Ostrowski † Bryan Rosenburg † Robert W. Wisniewski † Jimi Xenidis †
Michael Stumm ‡ Ben Gamsa § Reza Azimi ¶ Raymond Fingas ¶
Adrian Tam ¶ David Tam ¶

Abstract

Taking a traditional operating system and making it scale well on shared memory multiprocessors is hard. Performance debugging a multiprocessor operating system is hard. Maintaining a scalable operating system is hard. We contend that designing a system from the ground up specifically for scalability can simplify these tasks considerably and allows for reasonably straightforward implementations that are scalable and maintainable.

We have designed the K42 operating system from the start specifically for shared-memory multiprocessor scalability. Key to dealing with the complexities associated with scalability are (i) an object-oriented structure that maximizes locality, (ii) a Clustered Object infrastructure that supports distributed implementations in a straightforward way, and (iii) a hot-swapping infrastructure that allows one implementation to be exchanged for another at run-time as demands on objects change. In this paper, we describe K42's design, analyze its scalability and performance using real workloads running on 24 processor commercial hardware, and show some of K42's design tradeoffs as they relate to scalability.

1 Introduction

Taking a traditional operating system and making it scale well to large numbers of processors for diverse sets of workloads is extremely hard. Designing for scalability often hurts uniprocessor performance, and fast uniprocessor solutions typically don't scale. Even after reasonable scalability is achieved, these systems tend to be fragile. An increase in the number of processors or changes in hardware characteristics can require substantial new effort. Even minor software upgrades can turn into large efforts: simple innocuous mistakes can cause performance and scalability to degrade severely, and performance debugging these systems is entirely non-trivial.

We contend that structuring the operating system from the ground up to fundamentally minimize sharing makes it possible to achieve good scalability in a relatively straightforward, albeit non-trivial, way. It is too difficult to retrofit an existing system to arbitrarily scale, rather scalability must be taken in to account from the beginning.

We have designed the K42 operating system for scalability and implemented it using novel structures to minimize any form of sharing. While K42 inherited some of the infrastructure and ideas required for scalability from Tornado, a previous experimental operating system [11], these structures have been refined and tested in the context of a complete system running on commercial hardware, where it was possible to measure the end effects of specific design choices on real workloads. The design and implementation of K42 is complete in the sense that it is self-hosting. K42 supports the 64-bit PowerPC Linux API/ABI sufficiently to allow a large number of Linux binaries to be run unmodified on K42.

Experiments measuring scalability indicate that the approach used in K42 is very promising. Figures 1–3 depict the scalability of K42 with three different benchmarks running on a 24-way shared memory multiprocessor. As a reference point, we also included in the graphs results of running the exact same benchmark binaries on Linux version 2.4.19. (The details of the experiments and the methodology used is described in Appendix A.) For both the SPEC Development Environment Test (SDET), and Parallel Make benchmarks, K42 scales well up to 24 processors. For the PostMark benchmark, the speedup on K42 is good, and better than on

† IBM T. J. Watson Research Center
‡ Univ of Toronto, Dept of Electrical and Computer Engineering
§ SOMA Networks, Inc.
¶ University of Toronto, Dept of Computer Science

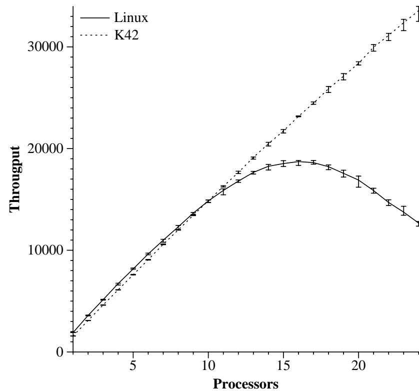

Figure 1: Throughput of SDET benchmark normalized to K42 uniprocessor result.

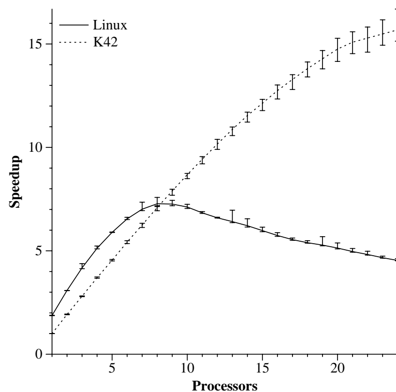

Figure 2: Speedup of p independent instances of Postmark normalized to K42 uniprocessor result.

Linux, but not yet perfect.[1]

In general, we observed that Linux scales reasonably well up to a certain number of processors, but its performance then begins to degrade for all but the Parallel Make benchmark. Linux scalability will likely continue to improve given the large number of contributions made each year. Nevertheless, K42, although implemented by a relatively small team, exhibits significantly better scalability characteristics under non-trivial workloads. We believe that new structures and techniques, such as those of K42, will become increasingly important as the the number of processors increase further and as newer hardware with increasing processor-memory speed disparity become more common.

We found three techniques to be especially important in achieving scalability:

[1]The bottleneck in this experiment is the file system memory allocator which does not yet have working support for large SMP or NUMA systems.

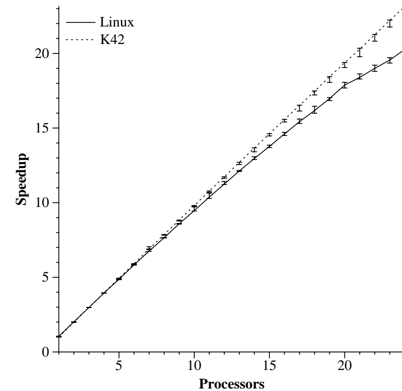

Figure 3: Speedup of p independent instances of Parallel Make normalized to K42 uniprocessor result.

1. Use of **object-oriented structure** and implementation. This allows K42 to avoid shared code paths, shared meta structures, such as global lookup tables, and global locks. Although having an object-oriented structure is often regarded as having overhead associated with it, in a multiprocessor environment, the object-oriented structure actually contributes to better performance. In K42, each virtual resource (e.g. VM region, network connection, file, process) and each physical resource (e.g. memory bank, network card, processor, disk) is managed by a different object instance. Each object instance encapsulates all the data necessary to manage the resource as well as all locks necessary to access the data. Hence, as long as application requests are to different resources, they are handled by the system entirely in parallel, with no shared data structures being traversed and no shared locks being accessed.
2. Use of **distribution and replication** in implementations of shared objects that are contended. When a resource instance is contended due to inherent sharing in the workload, performance can be significantly improved by using more complex, distributed implementations akin to those used in distributed systems, but designed for the characteristics of a shared memory multiprocessor. K42's **Clustered Object infrastructure** helps manage the complexity of distributed object implementations.[2] This infrastructure helps hide the distribution from the clients of the object, and it allows access to the object without requiring any remote or shared data access with a common case cost no higher than that of a virtual function call.
3. Use of **specialization**. Different resource instances may be represented by different object implemen-

[2]K42's Clustered Objects are similar Distributed Shared Objects [14] and Fragmented Objects[19, 4].

tations. At runtime, the object implementation is used that best matches the demands on the resource instance it represents. For example, a different file object implementation is used for small files than for large files. Additionally, different file implementations are used if a file is only open by one client versus being open concurrently by multiple clients. Doing so allows for more aggressive optimizations appropriate to the runtime characteristics of the resource instance in question. Specialization is also dynamic, whereby implementations are switched dynamically at run time using K42's **hot-swapping infrastructure**, when usage patterns change. Hence, when the file access pattern changes from being accessed by just a single thread to being accessed by many threads, then the implementation of the open file instance is dynamically switched from one optimized for single thread access to one optimized for shared access.

Just as important as these techniques is the major investment we made in core infrastructure that aid in maximizing locality. Examples of this are the Clustered Object infrastructure and the Hot-Swapping infrastructure mentioned above, but also a locality-preserving IPC facility, a locality-aware malloc subsystem, an object destruction strategy and facility that eliminates the need for existence locks, and a tracing facility that supports deterministic tracing.

In the remainder of the paper, we describe and analyze the techniques used to achieve scalability in K42. Section 2 first motivates many of the design choices by showing how false sharing alone can degrade performance by several orders of magnitude. Section 3 gives an overview of K42. Section 4 describes interesting examples of distributed object implementations used in K42 and their effect on performance. We close the paper with a review of related work and concluding remarks.

2 Motivation

To illustrate the magnitude of the performance impact of contention and sharing, consider the following experiment: each processor continuously, in a tight loop, issues a request to a server. The IPC between client and server and the request at the server is processed entirely on the same processor from which the request was issued, and no shared data needs to be accessed for the IPC. On the target hardware, the round trip IPC costs 1193 cycles. In involves an address space switch, transfer of several arguments, and authentication on both the send and reply path.[3] The increment of a variable in the uncontended case adds a single cycle to this number. Figure 4 shows the performance of 4 variants of this experiment, measuring the number of cycles needed for each round-trip request-response transaction:

1. Increment counter protected by lock: each request is to increment a shared counter, where a lock is acquired before the increment. This variant is represented by the top-most curve: at 24 processors, each transaction is slowed down by about a factor of about 19.
2. Increment counter using an atomic increment operation. This variant shows a steady degradation where at 24 processors, each request-response transaction is slowed down by a factor of about 12.
3. Increment per-processor counter in array. This variant has no logical sharing, but exhibits false sharing since multiple counters cohabit a single cache line. In the worst case, each request-response transaction is slowed down by a factor of about 6.
4. Increment padded per-processor counter in array. This variant is represented by the bottom-most curve that is entirely flat, indicating good speedup: up to 24 request-response transactions can be processed in parallel without interfering with each other.

These experiments show that any form of sharing in a critical path can be extremely costly — a simple mistake can cause one to quickly fall off of a performance cliff. Even though the potentially shared operation is, in the sequential case, less than one tenth of one percent of the overhead of the experiment, it quickly dominates performance if it is in a critical path on a multiprocessor system. This kind of dramatic result strongly suggests that we must simplify the task of the developer as much as possible, providing her with abstractions and infrastructure that simplifies the development of operating system code that minimizes sharing.

3 K42's scalability design

3.1 K42 Background

K42 employs a microkernel-based design. The microkernel provides memory management, process management, IPC, base scheduling, networking and device support. System servers include an NFS file server, name

[3] The cost for an IPC to the kernel, where no context switch is required, is 721 cycles.

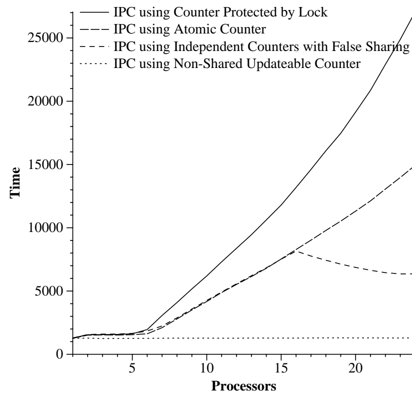

Figure 4: K42 microbenchmark measuring time for parallel client-server IPC request to update counter. Locks, shared data access and falsely shared cache lines all independently increase round-trip times significantly.

server, socket server, pty server and pipe server. Some of the functionality traditionally implemented in the kernel or servers is moved to libraries in the application's own address space in a fashion similar to that of Exokernel [9]. For example, thread scheduling is done by a user-level scheduler linked into each process.

All layers of K42, the kernel, system servers, and user-level libraries, make extensive use of object-oriented structures. All IPC is between objects in the client and server address spaces. We use a *stub compiler* with decorations on the C++ class declarations to automatically generate IPC calls from a client to a server, and have optimized these IPC paths to have good performance. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls.

Support of the Linux API is achieved with an emulation layer within each process that implements Linux system calls by method invocations on K42 objects. The emulation layer is reached by a K42-targeted version of the GNU C Library which is dynamically linked to the application (identical to PowerPC64 Linux version, except for system-call invocations). This approach has allowed us to run identical binaries on K42 and Linux for our experiments (except for the minor differences in C Libraries).

In this paper, we focus only on those aspects of K42 that relate to scalability. K42 is designed to run on systems with thousands of processors and support applications that may span the entire system, but also to run sequential and small-scale parallel applications as efficiently as on a small-scale multiprocessor.

3.2 Avoiding global locks and meta structures

K42 is designed to avoid any use of global locks or global meta structures, such as the call switch table, device switch table, global page caches, process table or file table used in some traditional operating systems. Instead K42 (i) uses an object-oriented structure which allows a decomposition that avoids sharing in the common case, and (ii) generally uses algorithms and implementations that do not rely on global knowledge or data. In particular, each physical and virtual resource is represented by an object, and for each resource instance there exists a separate object instance representing it. Each instance entirely encapsulates the data needed to manage the resource and the lock required to protect the data. Some examples of objects from the K42 kernel are:

- Process Objects: for each running process there is an instance of a Process Object in the kernel.
- Region Objects: for each binding of an address space region to a file region, a Region Object instance is created and attached to the Process Object.
- File Cache Manager (FCM) Objects: for each open file there is a separate FCM instance that manages the resident pages of the file.
- File Representative Objects: for each open file there is a File Representative Object instance capable of communicating with the file server on which the file exists.

In such an OO structured system, object handles directly represent relationships, as depicted in Figure 5. Independent OS requests tend to target different object instances, accessed by traversing independent interconnections between object instances. As a result, independent requests do not access shared data in the common case. Figure 6 abstractly compares the structure of an OO decomposed system to a more traditional system with its fast code paths and common switch tables and common data structures. On a uniprocessor, the traditional structure might even perform slightly better, but on a multiprocessor, the OO structure results in vastly better performance.

One consequence of maximizing locality is that it becomes more costly to have algorithms and policies decisions based on global information. As a result, localized decisions are made that are perhaps less optimized. An example (prevalent in many systems) is the dispatch queue: with a separate per-processor dispatch

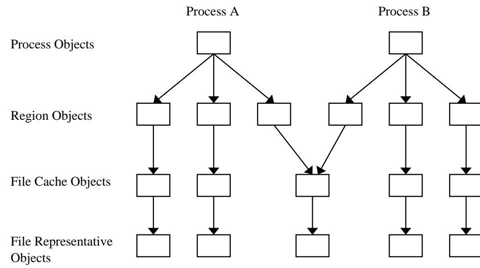

Figure 5: An illustration of the runtime instantiated network of objects in the K42 kernel representing two processes A and B both mapping 3 file into 3 regions of their address spaces, with one file being mapped into both.

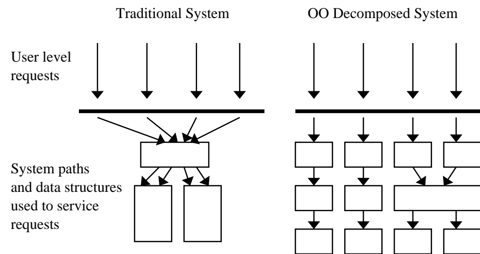

Figure 6: Contrast of the structure encouraged in a traditional system versus that of an Object Oriented one.

queue, it is no longer guaranteed that the n highest-priority threads system wide are executing at any given time. As another example, in K42, the global physical page manager does not manage all pages in the system. Instead, it allocates and reclaims pages to and from process-specific page managers who in turn allocate/reclaim pages to/from file-specific page managers. This localizes fine-grained decision making but prevents the implementation of many global policies, such as LRU. Finally, K42 uses per-processor page tables, allowing for a number of locality-enhancing optimizations. However, this comes at the cost of added complexity and overhead when migrating a process from one processor to another.

3.3 Clustered Objects

Although the OO structure of K42 can help reduce sharing by mapping independent resources to independent objects, some components, such as, say, the File Cache Manager (FCM) for a shared file, the Process object for a parallel program, or the system Page Manager, may be widely shared and hence require additional locality-enhancing measures. For these objects, K42 uses implementations that either partition and distribute the object across the processors/memory modules or that replicate read-mostly object data.

K42 uses a distributed component model called Clustered Objects to manage and hide the complexity of these distributed implementations. Clustered Objects are conceptually similar to design patterns such as Facade [10], and the partitioned object models used in Globe [14] and SOS [24]. However, Clustered Objects in K42 are specifically designed for shared-memory multiprocessors as opposed to loosely coupled distributed systems, and focus primarily on maximizing locality. From a client perspective, a Clustered Object appears as any other object: an OID is obtained on creation, and the OID is de-referenced to access the object. Hence, the client does not see the distribution within the implementation.

Internally, Clustered Objects are implemented as (possibly) multiple component objects each of which handles calls from a specified subset of processors, and a "root" object. Each component object represents the collective whole for some set of processors, and is called a representative. With the Clustered Object infrastructure, calls to the Clustered Object are automatically directed to the representative assigned to the processor from which the call is being made. The representatives must coordinate amongst themselves to ensure the Clustered Object externally appears consistent. Besides the representatives, each Clustered Object also has a root object, which is used to manage the global, shared data of the Clustered Object, and help coordinate activities between representatives.

The number of representatives used to implement the Clustered Object is a parameter of the Clustered Object. Clustered Objects in the current version of K42 either have a representative for each processor or are non-distributed with requests from all processors going to the root object acting as the single representative.[4] For Clustered Objects with multiple representatives, the representatives are typically created on demand when first accessed so as not to incur extra overhead when not needed.

Section 4 gives specific examples of distributed implementations using clustered objects. Details on the design and implementation of the Clustered Object infrastructure can be found in [11].

[4] The requests to the Clustered Object are still executed on the processor on which the request was issued, but data being accessed is typically shared.

3.4 Specialization and Hot-Swapping

Because K42 has each resource instance implemented by an independent object instance, resource management policies and implementations can be controlled on a per-resource basis. Thus, most K42 objects have multiple implementations, and the client or system chooses the best implementation at run time. This allows, for example, every open file to have a different pre-fetching policy, every memory region to have a different page size, and every process to have a different exception handling policy.

One of the lessons learned from Tornado and early stages of K42 was that specialization at instantiation time was too static, and that it was necessary to be able to dynamically change implementations at run time. For example, when a file is first opened by a thread, it may be the only thread accessing the file and hence it would make sense to choose an implementation optimized for single-thread access. However, if and when additional threads also open the same file, then the originally chosen implementation may perform poorly under the circumstances and a parallel implementation may be more suitable. For this reason, we have designed and implemented a dynamic object switching facility that allows object instances to be "hot-swapped" at run time, even while the object is being used [1, 2, 15, 25].

Thus, for sequential and small-scale parallel applications, implementations of resources can be used that have low overhead but do not scale, but as an application creates more threads, the system can hot-swap in implementations that can handle the new demands. Hot-swapping is facilitated in K42 by having each object be a Clustered Object, whether it has a distributed implementation or not. This adds one level of indirection with the attendant extra overhead for all calls, but gives us the flexibility to customize on the fly. (It also allows the interpositioning of objects, say for monitoring purposes.)

3.5 Core infrastructure supporting locality

In the development of K42, considerable effort went into designing and implementing core infrastructure to support locality. Key among these are the above-mentioned Clustered Object infrastructure, described in [11] and the Hot-swapping infrastructure, described in [25, 1]. In addition, we also found the following infrastructure to be critical:

- **Locality-aware memory allocator**: Using a design similar to that of [11], our allocator manages pools of per processor memory but also minimizes false sharing by properly padding allocated memory. It also provides NUMA support, although this does not come into play in the experiments presented here. The memory allocator itself is, of course, implemented so that it maximizes locality in its memory accesses and avoids global locks.
- **Locality-preserving interprocess communication**: The IPC mechanism of K42 is designed as a protected procedure call (PPC) between address spaces, with a design similar to the IPC mechanism of L4 [18]. The PPC facility is carefully crafted to ensure that a request from one address space to another (whether to the kernel or system server) is serviced on the same physical processor using efficient hand-off scheduling, maximizing memory locality, avoiding expensive cross processor interrupts, and providing as much concurrency as there are processors. Details on the implementation of our PPC facility can be found in [11, 17].
- **An object destruction strategy** that defers object destruction until all currently running threads have finished[5]. By deferring object destruction this way, any object can be safely accessed, even as the object i s being deleted. As a result, existence locks are no longer required, eliminating the need for most lock hierarchies. This in turn results in locks typically being held in sequence, significantly reducing lock hold times, and eliminating the need for complex deadlock avoidance algorithms. The implementation of our object destruction facility is described in [11].
- **A per-processor tracing facility**. Global trace buffers prevent deterministic gathering of trace information, in part because of locking requirements. By using per-processor trace buffers, non-blocking synchronization allows all local threads to share the buffer, allowing for deterministic gathering of trace data in a separate merge phase a posteriori. (Our tracing technology has been influential in the design of similar facilities for Linux.)

4 Distributed Data Structures

In this section we present three sample objects in K42 and describe how they can be distributed using the clustered object infrastructure. In two of the cases, we discuss how hot-swapping is used. We show how the distributed implementations result in major performance

[5] K42 is preemptable and has been designed for the general use of RCU techniques by ensuring that all system requests are handled on short-lived system threads.

gains. We also show how our design has allowed us to iteratively and incrementally introduce distribution in these objects, which illustrates a software engineering benefit clustered objects provide us.

4.1 Process Object

The Process Object represents a running process and all per-process operations are directed through it. For example, every page fault incurred by a process is directed to its Process Object for handling.

The Process Object maintains address space mappings as a list of Region Objects. When a page fault occurs, it searches its list in order to direct the fault to the appropriate Region Object. The left hand side of figure 7 illustrates the default non-distributed implementation. A single linked list with an associated lock is used to maintain the Region List. To ensure correctness in the face of concurrent access, an associated lock is acquired on traversals and modifications of the list.

In the non-distributed implementation, the list and its associated lock can become a bottleneck in the face of concurrent faults (see figure 8). As the number of concurrent threads is increased, the likelihood of the lock becoming contended grows, which can result in dramatic performance dropoffs. Even if the lock is not contended, then the read-write sharing of the cache line holding the lock and potential remote memory accesses for the region list elements add significant overhead.

We address this problem with a distributed Process object that caches the region list elements in a per-processor representative. (see right hand side of figure 7.) A master list, identical to the list maintained in the non-distributed version, is maintained in the root. When a fault occurs, the cache of the region list in the local representative is first consulted, acquiring only the local lock for uniprocessor correctness. If the region is not found there, the master list in the root is consulted and the result cached in the local list, acquiring and releasing the locks appropriately to ensure the required atomicity. This approach ensures that in general, the most common operation (looking up a region) will only access memory local to the processor and not require any inter-processor synchronization/communication.

It should be noted, however, that the distributed object has greater costs associated with region attachment and removal than the non-distributed implementation. When a new region is mapped, the new Region Object is first added to the master list in the root and then cached in the local list of the processor mapping the region. To un-map a region, its Region Object must atomically be found and removed first from the root and then all representatives that are caching the mapping. A lookup for a region not present in the local cache requires multiple searches and additional work to establish the mapping in the local list. In the case of a multi-threaded process, the overhead of the distributed implementation for region attachment and removal is more than made up for by the more efficient and frequent lookup operations.

In the case of a single-threaded application, all faults occur on a single processor and the distributed version provides no benefit, resulting in additional overheads both in terms of space and time. Hence, in order to maximize performance, the non-distributed Process Object is used by default. The distributed implementation is automatically switched to when a process becomes multithreaded using K42's hot-swapping facility.

Although one can imagine much more complex schemes for implementing distributed versions of the Process Object, the simple scheme chosen has three main advantages:

1. its performance characteristics are clear and easy to understand,
2. it preserves the majority of function of the non-distributed version making the implementation also easy to understand and maintain,
3. the distributed logic integrates easily into the non-distributed behaviour, relying on a simple model of caching which is implemented in a straightforward manner on top of the pre-existing data structures and facilities: a simple list which uses a single lock (identical to the list used in the non-distributed version) and the standard clustered object infrastructure for locating data members of the root object and the ability to iterate over the representatives.

4.2 Global Page Manager Object

K42 uses a number of distributed objects within the memory subsystem. One object of particular note is the Global Page Manager Object, the object at the root a hierarchical configuration of co-operating Page Management Objects (PM's). It is responsible for page management across all address spaces in the system[6]. File Cache Management Objects (FCM's) maintain the in-core pages of a file[7] and are attached to a Page Man-

[6] K42, employs a working set page management strategy.

[7] All memory regions of an address space in K42 are mapped by Region Objects to a specific file via an FCM. This includes explicitly

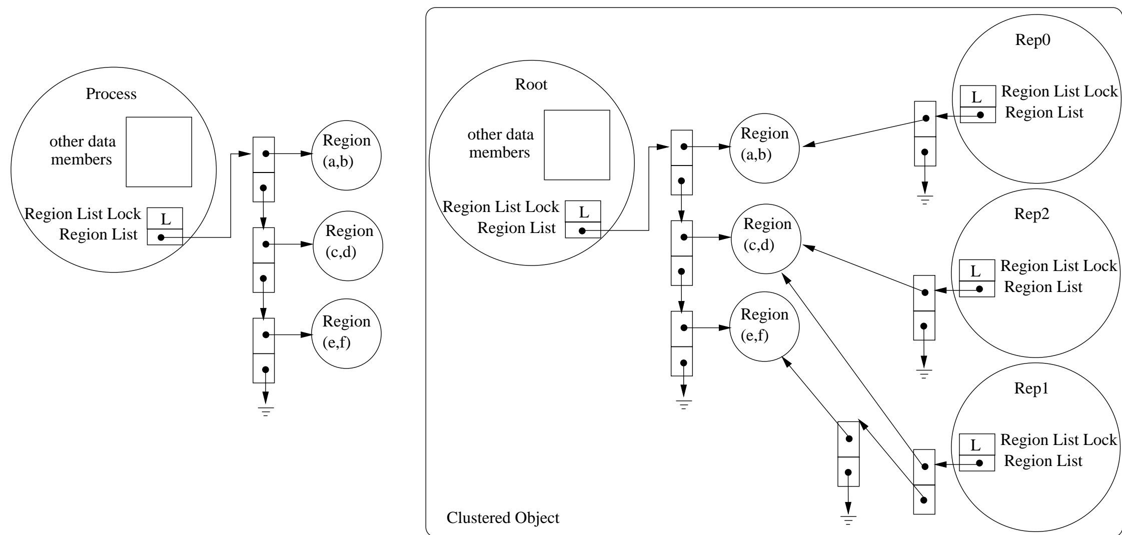

Figure 7: Non-Distributed and Distributed Process Objects

ager Object, from which they allocate and deallocate physical pages. FCM's for files not currently open by a process are attached to the Global Page Manager. The Global Page Manager implements reclamation by requesting pages back from the FCM's attached to it and from the Page Managers below it. Each Page Manager below the Global Page Manager is attached to a Process Object and implements page management for the open files associated with the process.

The left hand side of figure 9 illustrates the simple non-distributed implementation of the Global Page Manager that was first used in K42. It contains a free list of physical pages and two hash tables to record the attached FCM's and PM's. All three data structures were protected by a shared lock. On allocation and deallocation requests, the lock was acquired and the free list manipulated. Similarly when a PM or FCM was attached or removed, the lock was acquired and the appropriate hash table updated. Page usage statistics were maintained for each attached PM and FCM in the hash tables along with its object handle. Reclamation was implemented as locked iterations over the FCM's and PM's in the hash tables. Each FCM and PM was instructed to give back some number of pages based on its usage statistics during the reclamation iterations.

As the system matured, we progressively distributed the Global Page Manager Object in order to alleviate contention observed on the single lock and shared data structures. The current implementation is illustrated on the right hand side of figure 9. The first change was to introduce multiple representatives and maintain the free lists on a per-processor basis. The next change was to partition the FCM Hash Table on a per-processor basis by placing a separate FCM Hash Table into each representative and efficiently mapping an arbitrary FCM to a representative. The Clustered Object infrastructure provides a simple mapping of an object handle to the processor number on which the object was allocated. In the case of the distributed Global Page Manager, the allocating processor for an FCM is treated as its home processor which we map directly to a representative via an array of representatives maintained in the root. The FCM is stored in the hash table of representative of the home processor. The final change was to use the same approach to distribute the PM hash table.

In the distributed version, page allocations and deallocations are done on a per-processor basis by consulting only the per-representative free list. The current implementation does not attempt to balance the free lists across representatives. FCM and PM attachment and detachment are achieved by first calculating the home processor for the FCM or PM making the request and then placing it in the hash table of the representative associated with that processor. Given that there is only one instance of the Global Page Manager in the system and that there is a representative for every processor, an array of representative pointers was put into the root of the distributed Global Page Manager to facilitate more efficient mapping of processor to representative. The basic clustered object infrastructure provides facilities for locating

named files such as the program executable, as well as the anonymous files associated computational regions associated with a process such as its heap.

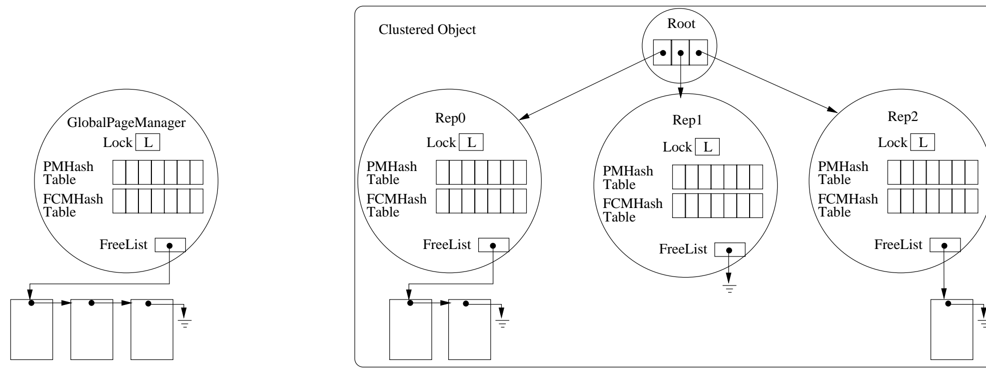

Figure 9: Non-Distributed vs Distributed Global Page Manager

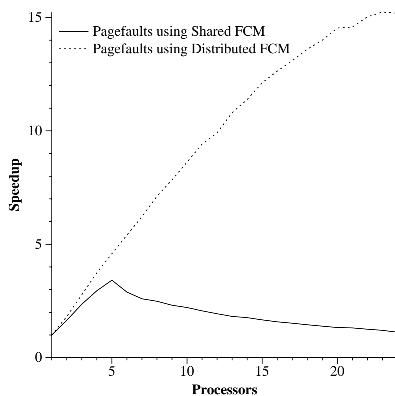

Figure 8: Graph of Distributed vs Non-Distributed Process Object: This graph illustrates an experiment showing the speedup of running a single process which spawns 1 thread per processor, each of which repeatedly maps an independent file into an independent region, accesses each page of the file sequentially, and then un-maps the region, measuring the average time for a thread to complete. The distributed implementation does not achieve perfect scalability because of idiosyncrasies in the TLBIE operation in the PowerPC hardware. The lock on the region list of the non-distributed implementation becomes a bottleneck rapidly, resulting in substantial slow down with more than 5 processors.

a representative, given a processor number, but it is designed for general use and is thus more costly. Reclamation is done completely local to a representative, with each representative iterating over the PM's and FCM's recorded in its hash tables.

The current distributed version of the Global Page Manager, in addition to alleviating the contention on the shared lock, in general eliminated the need for interprocessor communication. The array of representatives is the only shared state for the current Global Page Manager. It requires no locks and is read only after initialization and hence has good SMP cache performance. Figure 10 shows the impact the distributed Global Page Manager has on SDET performance.

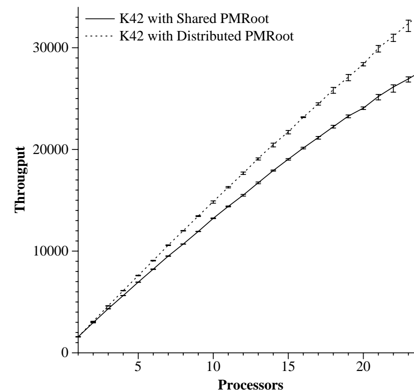

Figure 10: Graph of Distributed vs Non-Distributed Global Page Manager: This graph illustrates an experiment showing the speedup obtained running the SDET (workload described in the appendix, with results normalized to distributed uniprocessor result). with either the current distributed version of the Global Page Manager verse the original non-distributed version. The advantages of the distributed version become apparent even on a small number of processors.

Our research into distributed page management is not complete, but given our iterative approach, we are now in a better position to design new distributed implementations of the Global Page Manager which can be tested under real workloads. Perhaps this is one of the best-case examples of iterative clustered object development: a fully distributed implementation was developed by successively distributing a centralized implementation requiring minimal programming effort and with the centralized algorithms naturally being applied to the distributed version.

4.3 File Cache Managers

All regions of an address space are attached to an instance of a File Cache Manager (FCM) which caches the pages for that region. An FCM may be backed by a named file, or swap space in the case of anonymous regions, such as the heap of a process. An FCM is responsible for all aspects of in-core page management for the file it caches. On a page fualt, a Region Object asks its FCM to translate a file offset to a physical page frame. The translation may involve the allocation of new physical pages and the initiation of requests to a file system for the data. When a Page Manager asks it to give back pages, an FCM must implement local page reclamation over the pages it caches. The FCM is a complex object, implementing a number of intricate synchronization protocols including:

1. race-free page mapping and un-mapping,
2. asynchronous I/O between the faulting process and the file system,
3. timely and efficient page reclamation, and
4. maintenance of fork logic in the case of anonymous files.

The standard non-distributed FCM uses a single lock to ease the complexity of its internal implementation. When a file is accessed by a single process, the lock and centralized data structures do not pose a problem. When many processes or threads of a single process access the file concurrently, however, then the shared lock and data structures induce inter-processor communication resulting in degraded page fault performance.

Unlike the Process Object and Global Page Manager, there is no straightforward way to distribute the FCM's data members without adding considerable complexity and breaking its internal protocols. Rather than redesigning every one of its internal operations, a new distributed version was developed by replacing the core lookup hash table with a reusable encapsulated distributed hash table. This allowed the majority of the protocols to be preserved while optimizing the critical page lookup paths in an isolated fashion. Moreover, it allowed us to reuse the distributed hash table in other objects.

Figure 11 illustrates the basic structure of the reusable distributed hash table (DHash). There are two basic components to the DHash: a MasterDHashTable and LocalDHashTables, which are designed to be embedded into a Clustered Object's root and representatives respectively. After embedding the DHash into a Clustered Object, calls can be made to either the LocalDHashTables or the MasterDHashTable directly. DHash has a number of interesting features:

1. LocalDHashTables and MasterDHashTable automatically cooperate to provide the semantics of a single shared hash table for common operations, hiding its internal complexity,

2. all locking and synchronization are handled internally,

3. all tables automatically and independently size themselves,

4. uses fine-grain locking where common accesses require an increment of a reference count and the locking of a single target data element,

5. supports data elements which have both shared and distributed constituents,

6. supports scatter and gather operations for distributed data elements

The support for distributed data was used to aggressively optimize the distributed FCM. A developer can specify that the data to be stored in the DHash is composed of global data values as well as local data values. When querying a LocalDHashTable, the local data values will be returned along with a local copy of the global values. The MasterDHashTable stores the primary version of global values associated with the key. Scatter and gather operators allow the developer to efficiently aggregate and update the distributed data members. In the case of the distributed FCM, a DHash table is used to store the page descriptors associated with each page cached. By distributing the page descriptor's state bits, it was possible to implement concurrent faults to a single logical page descriptor which requires no synchronization or communication. For example: the dirty bit for a page must be updated when a write fault initially occurs on a page. By distributing the dirty bit each processor need only examine and update its own version of the bit, not requiring synchronization. During page scanning however, the global value of the dirty bit requires a gather operation.

If not under contention the distributed FCM has overhead associated both in terms of time and space. The distributed FCM essentially has double the space requirements due to the maintenance of both the local and master portions of the DHash table. By default we use the non-distributed FCM when a file is opened, if the file suffers contention we swap to the distributed implementation.

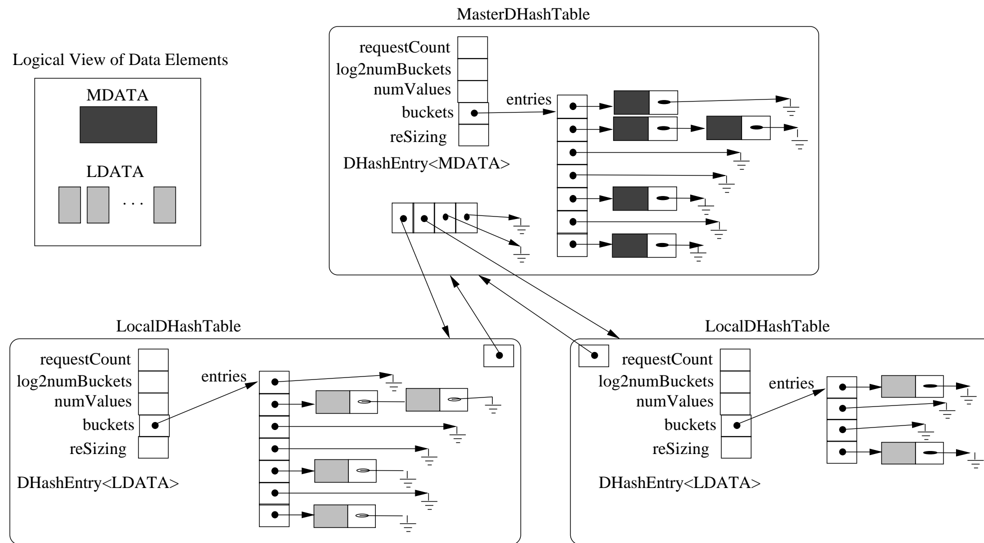

Figure 11: Reusable Distributed Hash Table Clustered Object Components

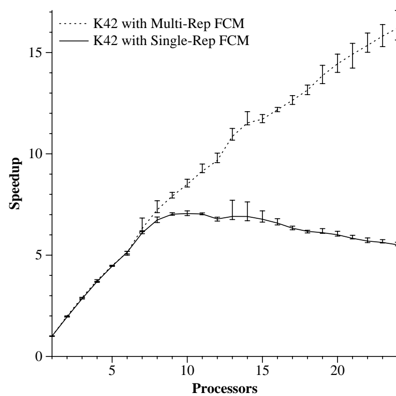

Figure 12: Graph of Distributed FCM vs Non-Distributed FCM performance: This graph illustrates the results of running one instance of grep per processor. Each grep searched a common 111MB file for a non-matching search pattern. We measured the average time for an instance of grep to complete. Although not visible here, the non-distributed version has approximately a 1% advantage when running on 1 processor.

4.4 Experiences

We found that the notion of a "root" in a clustered object is critical for a number of reasons. First, it gives us a natural strategy to incrementally distribute an implementation, where the root is the original non-distributed implementation and we distribute just the function that is hot. Second, it enables a natural implementation for what we have found to be a common case, i.e., the reps just implement a cache of state while there is a centralized implementation to handle all the modifications. Finally, making the root a standard part of our infrastructure has allowed us to provide standard mechanisms for maintaining information like the list of all representatives and a place to stand for synchronization when destroying objects. Our experience is that it is critical to put all this complexity in the infrastructure rather than burden programmers of individual objects with it.

When we first obtained our 24-processor system we found that our scalability was terrible. We were able to achieve a good level of scalability in about two weeks of work. The five main reasons we were able to accomplish this were:

1. While we had not experimented on a large system, the design had been structured for good scalability from the start.
2. The OO design meant that all the changes were encapsulated within objects. There were no cases where changes to a data structure resulted in changes propagating to other parts of the system.
3. The deferred deletion (i.e. RCU), eliminates most of the existence locks and hence locking hierarchies. We have found no cases where we had to modify complex locking protocols that permeated multiple objects.
4. The OO design and hot-swapping allowed us to develop special purpose objects to deal with some of the hard cases without polluting our common case objects.
5. We had invested a major effort in tracing infrastructure that allowed us to easily pinpoint performance problems and fix them.

It appears that the OO design has so far given us a major software engineering advantage. We have found that it is much easier to write multiple simple special-purpose objects rather than the normal strategy of having a single complex implementation of a system service that handles all the special cases. However, we do have a long-term concern. As more and more implementations are developed, interface changes and bug fixes might result in a maintenance nightmare. While we use inheritance aggressively in the system, in many cases (e.g., non-distributed versus distributed implementations) code cannot be shared easily. We are looking at various technical solutions to this problem.

Another challenge is that it can be difficult to achieve a global state if all the data for achieving that understanding is scattered throughout many object instances. For example, without a single page cache, there is no natural way to implement a global clock algorithm. We have so far found that alternative algorithms (e.g., working set) are feasible.

5 Related work

Rosenblum et al in [8] analyzes the difficulties in dealing with coherency overheads induced by communication and synchronization, and the importance of avoiding false sharing and preserving cache locality in achieving high performance on SMMPs.

Many techniques have been developed to address synchronization and locality problems in multiprocessor environments. The communication scalability of synchronization has been improved by the introduction of distributed, queue-based spin-locks[22]. Hoard[3] and the Sequent Allocator[21] employ a combination of global heap and per-processor heaps to achieve scalable memory allocation and avoid cache thrashing; K42's memory allocator follows the same principles. Papers such as [6, 20, 23] describe approaches to multiprocessor performance issues in the context of specific subsystems. The work involved splitting specific locks and changing specific data structures to avoid cache collision.

Hive[8] is an operating system designed to achieve scalability and reliability. It is structured as an internal distributed system of independent kernels, and it relies on write-protection hardware. Disco[5] proposed using virtual machines to provide scalability and to hide some of the characteristics of the underlying hardware from the NUMA-unaware operating system. Cellular Disco[12] extends Disco to support hardware fault containment.

Spring[13] and Choices[7] are object-oriented operating systems targeting distributed environments. Spring also pursued customization of services (the notion of *subcontracts*, but K42's hot-swapping approach allows for more flexibility.

SOS[24] and Globe[14] present concepts similar to clustered objects, but focusing on distributed environments, which have more complex failure modes and very different (less constrained) efficiency requirements than a tightly coupled shared-memory multiprocessor environment.

6 Concluding Remarks

K42 was developed from scratch for shared memory multiprocessors. Key aspects of the design are: (i) an object-oriented structure that maximizes locality, (ii) a Clustered-Object infrastructure that support distributed implementations in a straightforward way, and (iii) a hot-swapping infrastructure that allows one implementation to be exchanged for another at run-time as demands on objects change. We demonstrated excellent scalability for three system benchmarks, described the basic system infrastructure, and presented three examples of clustered objects. For these clustered objects, we discussed how we distributed their implementation, and showed substantial impact on performance. It appears that the basic ideas we have explored work reasonably well.

We have been developing K42 as a research vehicle for the last five years in a major industry lab by, on average a team of six researchers and a number of academic collaborators. K42's design has borrowed heavily from the Tornado OS previously developed at the University of Toronto. K42's support of the 64-bit PowerPC Linux APIs and ABI allows it to exploit Linux's rich software ecosystem (and work is underway on an x86-64 port). The system is available under an LGPL license.

While the investment in K42 is large by the standards of research projects, it is very small by the standards of fully functional operating systems. As we have seen in this paper, K42 is mature enough to run real applications and benchmarks. Substantial subsystems, like Apache, run without modification or re-compilation from 64-bit PowerPC Linux. Services like shared libraries, pty support, NFS, job control, and much of the less exciting but tough and important function normally not fully supported in research environments is fully supported in K42.

We have found K42's OO-design to be ideal for prototyping new technologies and ideas, and as such this project has had significant impact on Linux. The ABI and toolchain for 64-bit PowerPC Linux, deferred object deletion technology (Read-Copy Update), the Linux Tracing Toolkit and reverse-mapping logic for the Linux 2.5/2.6 VM are examples of areas where experience and technology from K42 has made an impact on Linux. Our experience is that K42 is a great proving ground for technology and once an idea has been demonstrated in K42, it is often possible to transfer it to Linux.

With K42's OO design and user-level library implementation of system services, basic efficiency and uniprocessor performance has been a worry. Our concerns are somewhat alleviated by the current performance numbers. Our uniprocessor degradation is, for OS intensive load, less than 10% over Linux. We have found that most of the performance challenges have come from the user-level library implementation of services and not from the OO design. We are actively working on base performance, and believe that for most applications and workloads we will be able to match and in some cases even exceed Linux's uniprocessor performance in the near future. Our dependency on 64-bits means that we won't be applicable to many current low end systems.

A Methodology and Details of Experiments Run

All the results in this paper were obtained by running K42 or Linux 2.4.19 on PowerPC hardware. We used an S85 Enterprise Server IBM RS/6000 PowerPC bus-based cache-coherent multiprocessor with 24 600MHZ RS64-IV processors and 16GB of main memory. The three different benchmarks we ran to examine scalability were SPEC SDET, Postmark 1.5, and a parallel make. To stress the system rather than disk, each of the experiments was run using RamFS. For each experiment we ran the same script on both K42 and Linux as distributed by SuSE (with the O(1) scheduler patch).

The SPEC Software Development Environment Throughput (SDET) benchmark [26] consists of a script that executes a series common Unix commands and programs including ls, nroff, gcc, grep, etc. Each of these are run in sequence. For our experiemnts, the SDET benchmark was modified by removing some system utilities such as "ps" and "df". To examine scalability we ran one script per processor. All the user programs (bash, gcc, ls, etc.) are the exact same binary, whether run on K42 or Linux. The same version of glibc 2.2.5 was used, but modified on K42 to intercept and direct the system calls to the K42 implementations. The throughput numbers are those reported by the SDET benchmark and represent the number of scripts per hour that are executed.

Postmark was designed to model a combination of electronic mail, netnews, and web-based commerce transactions [16]. It creates a large number of small, randomly-sized files and performs a specified number of transactions on them. Each transaction consists of a randomly chosen pairing of file creation or deletion with file read or append. As with SDET, a separated instance of Postmark was created for each processor with corresponding separate directories. We ran the benchmark with 20,000 files, 100,000 transaction and disabled Unix buffered files. The rest of the options were the default. The total time reported is obtained by summing the time each individual instance takes.

Parallel make is designed to model the common task of building an application in parallel (in this case the randomly chosen application is GNU Flex). Rather than invoking "make" with a "-j" option telling it to issue commands in parallel, we created one build directory per processor and invoked one sequential "make" in each of these directories in parallel (all "make"'s built from a common source tree, to unique build directories). A driver application synchronized the invocations of each make process to ensure simultaneous start times, tracked the run-time of each "make" and reported the final result as an average of all "make" run-times. GNU Make 3.79 and GCC 3.2.2 were used.

References

[1] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[2] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert Wisniewski, Dilma da Silva, Orran Krieger, and Craig Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, 2002.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable

memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

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

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[6] M. Campbell, R. Barton, J. Browning, B. Curry, T. Davis, T. Edmonds, R. Hold, J. Slice, T. Smith, and R. Wescott. The parallelizatin of UNIX system v release 4.0. In *Proc. USENIX Technical Conference*, pages 307–324, January 1991.

[7] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[8] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosio, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, 1995.

[9] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating System Principles*, volume 29, 3–6 December 1995.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patters Elements of Reuseable Object-Oriented Software*. Addison Wesley, 1994.

[11] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, 22-25 February 1999.

[12] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 18(3):229–262, 2000.

[13] Graham Hamilton and Panos Kougiouris. The spring nucleus: A microkernel for objects. In *Summer USENIX Conference*, pages 147–160, June 1993.

[14] P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and W. de Jonge. An object model for flexible distributed systems. In *First Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, may 1995.

[15] Kevin Hui, Jonathan Appavoo, Robert Wisniewski, Marc Auslander, David Edelsohn, Ben Gamsa, Orran Krieger, Bryan Rosenburg, and Michael Stumm. Position summary: Supporting hot-swappable components for system software. In *HotOS*, 2001.

[16] Jeffrey Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance. PostMark: A New File System Benchmark.

[17] O. Krieger, M. Stumm, and R. Unrau. A fair fast scalable reader-writer lock. In *PROC of the 1993 ICPP*, St. Charles, IL, August 16-20 1993.

[18] J. Liedtke, K. Elphinstone, S. Schoenberg, and H. Haertig. Achieved IPC performance. In IEEE, editor, *The Sixth Workshop on Hot Topics in Operating Systems: May 5–6, 1997, Cape Cod, Massachusetts*, pages 28–31, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. IEEE Computer Society Press.

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

[20] D. McCrocklin. Scaling solaris for enterprise computing. In *Spring 1995 Cray Users Group Meeting*, pages 172–181, 1995.

[21] Paul E. McKenney, Jack Slingwine, and Phil Krueger. Experience with an efficient parallel kernel memory allocator. *Software–Pratice and Experience*, 31(3):235–257, 2001.

[22] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *TOCS*, 9(1):pp. 21–65, February 1991.

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

[24] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. Sos: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[25] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX*, page to appear, San Antonio, TX, June 2003.

[26] spec.org. SPEC SDM suite. http://www.spec.org/osg/sdm91/, 1996.