Experience Distributing Objects in an SMMP OS

JONATHAN APPAVOO, DILMA DA SILVA, ORRAN KRIEGER,
MARC AUSLANDER, MICHAL OSTROWSKI, BRYAN ROSENBURG,
AMOS WATERLAND, ROBERT W. WISNIEWSKI, and JIMI XENIDIS
IBM T.J. Watson Research Center
and
MICHAEL STUMM and LIVIO SOARES
Dept. of Electrical and Computer Engineering, University of Toronto

Designing and implementing system software so that it scales well on shared-memory multiprocessors (SMMPs) has proven to be surprisingly challenging. To improve scalability, most designers to date have focused on concurrency by iteratively eliminating the need for locks and reducing lock contention. However, our experience indicates that locality is just as, if not more, important and that focusing on locality ultimately leads to a more scalable system.

In this paper, we describe a methodology and a framework for constructing system software structured for locality, exploiting techniques similar to those used in distributed systems. Specifically, we found two techniques to be effective in improving scalability of SMMP operating systems: (*i*) an object-oriented structure that minimizes sharing by providing a natural mapping from independent requests to independent code paths and data structures, and (*ii*) the selective partitioning, distribution, and replication of object implementations in order to improve locality. We describe concrete examples of distributed objects and our experience implementing them. We demonstrate that the distributed implementations improve the scalability of operating-system-intensive parallel workloads.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design

Additional Key Words and Phrases: Locality, Concurrency, Distribution, Scalability SMMP

ACM Reference Format:

Appavoo, J., da Silva, D., Krieger, O., Auslander, M., Ostrowski, M., Rosenburg, B., Waterland, A., Wisniewski, R. W., Xenidis, J., Stumm, M., and Soares L. Experience distributing objects

Authors' addresses: J. Appavoo, D. da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis; e-mail: {jappavoo,dilmasilva,okrieg,Marc_Auslander, mostrows,rosnbrg,apw,bobww,jimix}@us.ibm.com; M. Stumm and L. Soares: Department of Electrical and Computer Engineering, University of Toronto; e-mail: {stumm,livio}@eecg.toronto.edu

in an SMMP OS. ACM Trans. Comput. Syst. 25, 3, Article 6 (August 2007), 52 pages. DOI = 10.1145/1275517.1275518 http://doi.acm.org/10.1145/1275517.1275518

1. INTRODUCTION

On shared-memory multiprocessors (SMMPs), data accessed by only one processor is said to have good locality, while data accessed by many processors is said to have poor locality. Poor locality is typified by contended locks and a large number of cache misses due to shared-write data accesses. Maximizing locality on SMMPs is critical to achieving good performance and scalability. A single shared cache-line on a critical system code path can have a significant negative impact on performance and scalability.

In this article, we describe a methodology and a framework for constructing system software that is structured for locality. We have applied the methodology in designing an SMMP operating system called K42. The methodology and framework we have developed is the result of over 10 years experience implementing SMMP operating systems: first Hurricane [Unrau et al. 1995], then Tornado [Gamsa 1999] (both at the University of Toronto), and finally K42 jointly at IBM and the University of Toronto. While we have experience only with operating systems, the methodology should be equally relevant to other transaction-driven software, such as databases and Web servers, where parallelism results from parallel requests by the workload.

Techniques for improving scalability of SMMP operating systems are relevant, as demonstrated by the efforts that have gone into parallelizing Linux and commercial operating systems such as AIX, Solaris, and Irix over the last several years. An increasingly large number of commercial 64-way and even 128-way MPs are being used as web or database (DB) servers. Some next-generation game consoles are 4-way SMMPs. Given the trend in multicore, multithreaded chips, it is easy to envision 32- or 64-way SMMP desktop PCs in the not-too-distant future.

Traditionally, system designers have focused on concurrency rather than locality when developing and scaling SMMP operating systems. We argue that the approach generally used is ad hoc in nature and typically leads to complex implementations, while providing little flexibility. Adding more processors to the system, or changing access patterns, may require significant retuning.

In contrast, the central tenet of our work has been to primarily focus on locality by applying techniques similar to those used in distributed systems for selectively *partitioning*, *distributing*, and *replicating* data, using a framework we developed that supports locality optimizations within an object-oriented software model. Hence, we exploit object-oriented structures primarily for locality-centric performance optimization, and only secondarily for the software engineering benefits or for flexibility. The result, we argue, is a system with a simpler overall structure that greatly reduces the complexity required to achieve good scalability. The approach used allows the developer to focus on specific objects, without having to worry about larger system-wide protocols, enabling an incremental optimization strategy. Objects are specialized to handle specific

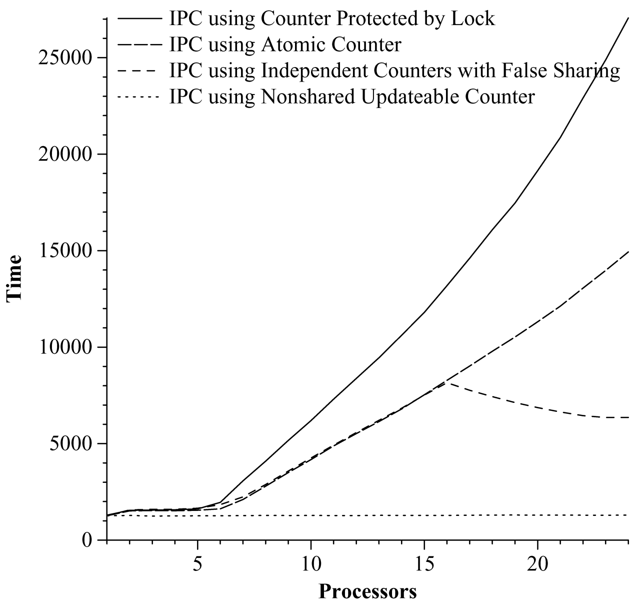

Fig. 1. K42 microbenchmark measuring cycles required for parallel client-server IPC requests to update a counter. Locks, shared data access, and falsely shared cache lines all increase round-trip times significantly even though the shared data access constitutes less than one tenth of one percent of the overhead in the uncontended case.

demands, leading to a separation of concerns and simplifying the development process.

1.1 Motivating Example

To illustrate the magnitude of the performance impact of contention and sharing in an SMMP, consider the following simple experiment: each processor continuously, in a tight loop, issues a request to a server. The interprocess communication (IPC) between client and server and the request-handling at the server are processed entirely on the 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.[1] It involves an address space switch, transfer of several arguments, and authentication on both the send and reply paths. The increment of a variable in the uncontended case adds a single cycle to this number.

Figure 1 shows the performance of 4 variants of this experiment, measuring the number of cycles needed for each round-trip request-response transaction when the number of participating processors is increased from 1 to 24:

(1) Increment counter protected by lock. This variant is represented by the topmost curve: at 24 processors, each transaction is slowed down by a factor of about 19.

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

(2) Increment counter using an atomic increment operation. This variant shows a steady degradation. 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 because 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 bottommost curve that is entirely flat, indicating good speedup: 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 a performance cliff.[2] 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 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.

1.2 Scalablity and Locality

An operating system is said to be scalable if it can utilize the processing, communication, and memory resources of the hardware such that adding resources permits a proportional increase in the load that the operating system can satisfy. Operating system load is considered to be the demand on the basic kernel services.

Software designed for locality on an SMMP uses partitioning, distribution, and replication of data to control concurrent access. Doing so provides fine-grain control over memory accesses and control over the associated communication. For example, using a distributed implementation of a performance counter, where each processor is assigned its own local padded subcounter, ensures that updates to the counter result only in local communication.

It is worth noting that there are two forms of communication associated with a memory access: (*i*) utilization of the interconnect fabric to bring the data from storage into the processor's caching system, and (*ii*) utilization of the interconnect for the associated cache coherency messages. On large-scale machines, with complex distributed interconnection networks, nonuniform memory access (NUMA) effects increase the benefit of localizing memory accesses by avoiding remote delays. Additionally, localizing memory accesses restricts communication to local portions of a complex hierarchical interconnect, and thereby avoids the use (and thus congestion) of the global portions of the memory interconnect.

[2]While this experiment may seem contrived, a number of examples we introduce in later sections show that this behavior can easily occur in practice.

Even small machines, with flat, high-capacity interconnects that do not suffer congestion delays, benefit from localizing memory accesses. The use of large multilayer caches to mitigate the disparities between processor frequencies and main memory response times results in NUMA-like behavior. Considerable delays result when a cache miss occurs. Partitioning, distribution, and replication help avoid cache misses associated with shared data access by reducing the use of shared data on critical paths. For example, a distributed performance counter with all subcounters residing on separate cache lines eliminates cache coherency actions (invalidations to ensure consistency) on updates, thus making communication delays associated with cache misses less likely. There is a tradeoff, however: when the value of the counter is needed, the values of the individual subcounters must be gathered and additional cache misses must be suffered.

Implementations utilizing partitioning, distribution, and replication require programmers to explicitly control and express how the data structures will grow with respect to memory consumption as additional load is observed. For example, consider a distributed table used to record processes, with an implementation that utilizes a table per processor that records just the processes created on that processor. In such an approach the OS programmer explicitly manages the required memory on a per-processor basis. This implementation will naturally adapt to the size of the machine on which it is running, and on large-scale machines with distributed memory banks, data growth can occur in a balanced fashion.

1.3 Overview

Fundamental to the work presented in this paper are K42's object-oriented structure and K42's support for Clustered Objects. Section 2 provides background on K42's structure. In Section 3, we identify the principles we have followed that allow the development of localized, scalable data structures. The three main points are:

(1) Focus on locality, not concurrency, to achieve good scalability;
(2) Adopt a model for distributed components to enable pervasive and consistent construction of locality-tuned implementations;
(3) Support distribution within an object-oriented encapsulation boundary to ease complexity and permit controlled and manageable introduction of localized data structures into complex systems software.

In Section 4, we present a case study of how we distribute objects used for virtual memory management in K42.[3] Despite the complexity associated with distribution, we demonstrate that a fully distributed implementation of this core service is possible. We further demonstrate that the techniques are affordable; uniprocessor performance competitive with traditional systems is achieved. For a multiuser workload on a 24-way system, the use of distributed memory-management objects results in a 70% improvement in performance

[3]We selected the memory management subsystem for this case study because (*i*) it is critical to the performance of many applications, and (*ii*) it represents a complex core service of the operating system. The same techniques can and have been applied to other subsystems.

as compared with a system using nondistributed implementations of the same objects, as we show in Section 5.

Our general approach is as follows. We start with simple implementations and then introduce distributed implementations as needed, with the goal of exploring our infrastructure and the feasibility of our approach. The goal is not to argue that the distributed implementations chosen are either optimal or even superior to equivalent highly-concurrent versions—rather it is to validate the methodology and gain insights into the techniques for constructing a system that can feasibly employ locality in a fine-grain fashion. For example, using a lock-free list instead of a set of distributed lists may provide equal performance for a specific machine and load. However, we take it as a basic axiom that locality-based implementations have the desirable properties of permitting explicit control of communication costs.

The approach we have taken does not preclude the use of highly concurrent structures. Rather we propose a metastructure that permits the designer and implementor of a system to mix and match centralized and distributed implementations.

Object-oriented encapsulation and support for distribution are our primary tools in the pursuit of locality optimizations. We integrate mechanisms and primitives for developing distributed implementations within a general software construction framework. A developer is free to use any technique to implement a given object—coarsely synchronized nondistributed data structures, highly concurrent nondistributed data structures, distributed data structures, or hybrids. With our support for distribution integrated in an accepted software engineering model, we are able to iteratively and incrementally introduce compatible implementations that vary in scalability.

Section 6 contains a comprehensive review of related work. Our conclusions are in Section 7. In many of the sections we include the design principles and lessons learned that pertain specifically to the content described in that section. Overall principles and experiences are presented in the introduction and conclusion.

2. K42 BACKGROUND

In this section, we present background material on the K42 operating system needed to understand the remainder of the paper.

Our implementation of K42 is PowerPC Linux API- and ABI-compatible. It supports the same application libraries as Linux and provides the same device driver and kernel module infrastructure, extensively exploiting Linux code to accomplish this [Appavoo et al. 2003]. In essence, K42 provides an alternative implementation of the core of the Linux kernel.

Fundamental to maximizing locality in K42 are (i) K42's object-oriented structure and (ii) K42's support for Clustered Objects. Some aspects of K42's core infrastructure that are important for achieving scalability are its locality-aware memory allocator, its locality-preserving interprocess communication mechanism, and its support for read-copy-update (RCU [McKenney et al. 2002]) object destruction.

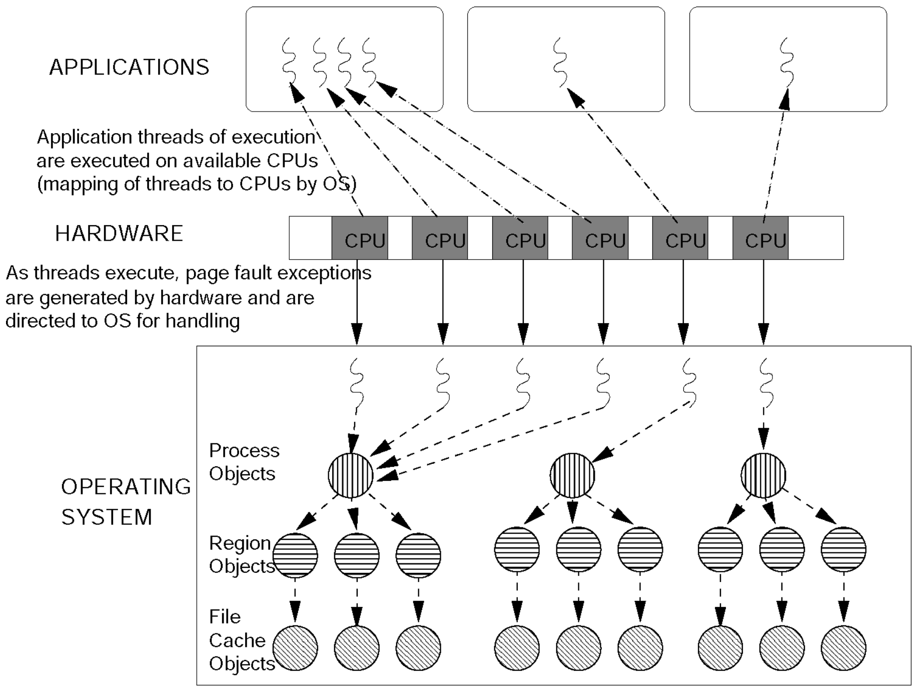
APPLICATIONS
Application threads of execution are executed on available CPUs (mapping of threads to CPUs by OS)
HARDWARE
CPU
CPU
CPU
CPU
CPU
CPU
As threads execute, page fault exceptions are generated by hardware and are directed to OS for handling
OPERATING SYSTEM
Process Objects
Region Objects
File Cache Objects

Fig. 2. An illustration of the K42 runtime structure used to handle in-core page faults.

2.1 K42's Object-Oriented Structure

K42 uses an object-oriented design that promotes a natural decomposition that avoids sharing in the common case. In K42, each physical and virtual resource instance is represented by an individual object instance. Each object instance encapsulates the data needed to manage the resource instance as well as the locks required to protect the data. Thus, K42 is designed to avoid global locks or global data structures, such as the process table, the file table, the global page cache, the call-switch table, or the device-switch table used in some traditional operating systems.

We illustrate our approach with a description of in-core page fault handling. We decompose the page fault handler execution into calls to independent objects as depicted in Figure 2. Each page fault is handled by an independent kernel thread that is scheduled on the processor on which the fault occurred. Execution of each kernel thread servicing a page fault proceeds on an independent path through the following objects associated with the user-level thread and the faulting page:

—**Process Objects**: For each running process there is an instance of a Process Object in the kernel. Kernel threads handling page faults for independent processes are directed to independent process objects, avoiding the need for common data accesses.

—**Region Objects**: For each binding of an address space region to a file region, a Region Object instance is attached to the Process Object. Threads within a

Table I. Kernel Object Accesses for Three User Scenarios (each entry details the accesses to a unique object instance. We include the objects that account for 80% of all accesses. The final entry in each column shows the number of objects needed to account for that 80%)

Multi-User		Single-User		Multi-Threaded	
Accesses (%)	*Object Type*	*Accesses (%)*	*Object Type*	*Accesses (%)*	*Object Type*
9569 (13.38)	Process *(P1)*	9569 (13.62)	Process*(P1)*	18928 (27.40)	Process
9365 (13.10)	Process *(P2)*	9569 (13.62)	Process *(P2)*	9372 (13.56)	HAT
4692 (6.56)	HAT *(P1)*	8196 (11.66)	FCMFile *(P1,P2)*	8239 (11.93)	SegmentHAT
4600 (6.43)	HAT *(P2)*	4692 (6.68)	HAT *(P1)*	8197 (11.86)	FCMFile
4138 (5.79)	SegmentHAT *(P1)*	4692 (6.68)	HAT *(P2)*	8196 (11.86)	Region
4136 (5.78)	SegmentHAT *(P2)*	4136 (5.89)	SegmentHAT *(P1)*	1249 (1.81)	FCMComp
4101 (5.73)	FCMFile *(P1)*	4136 (5.89)	SegmentHAT *(P2)*	1224 (1.77)	GlobalPM
4099 (5.73)	Region *(P1)*	4097 (5.83)	Region *(P1)*		
4098 (5.73)	FCMFile *(P2)*	4097 (5.83)	Region *(P2)*		
4098 (5.73)	Region *(P2)*	1181 (1.68)	GlobalPM *(P1,P2)*		
1145 (1.60)	GlobalPM *(P1,P2)*	853 (1.21)	FCMComp *(P1)*		
910 (1.27)	FCMComp *(P1)*	853 (1.21)	FCMComp *(P2)*		
840 (1.17)	FCMComp *(P2)*	447 (0.64)	PMLeaf *(P1,P2)*		
666 (0.93)	PageAllocator *(P1,P2)*				
503 (0.70)	COSMgr *(P1,P2)*				
471 (0.66)	Process *(P3)*				
57431 (80.31)	16	56518 (80.43)	13	55405 (80.19)	7

process that access different regions of an address space access independent Region Objects.

—**File Cache Manager (FCM) Objects**: For each open file there is a separate FCM instance that manages the resident pages of the file. Page faults for data from independent files are serviced by independent FCM Objects.

Like other research systems, K42 has an object-oriented design, but not primarily for the software engineering benefits or for flexibility, but rather for multiprocessor performance benefits. More specifically, the design of K42 was based on the observations that: (*i*) operating systems are driven by requests from applications for virtual resources; (*ii*) to achieve good performance on multiprocessors, requests to different resources should be handled independently, that is, without accessing any common data structures and without acquiring any common locks, and (*iii*) the requests should, in the common case, be serviced on the processors on which they are issued. As illustrated above, using in-core page fault handling as an example, K42 uses an object-oriented approach to yield a runtime structure in which execution paths for independent operating system requests, in this case page faults, access independent data structures, thus promoting locality.

Table I illustrates quantitatively the independence of objects within K42. In the table, we consider three different scenarios involving programs scanning a file:

(1) entirely independent requests to the OS: two users run different programs to scan independent files;

(2) partially independent requests to the OS: a single user runs two instances of the same program to scan the same file; and
(3) nonindependent requests to the OS: a user runs a multithreaded program with two threads scanning the same file.

Each row of the table indicates the number of accesses to a particular object and the percent of all accesses it represents. The table shows that the higher the degree of independence in requests, the larger the number of objects involved and the lower the amount of sharing. That is, the object-oriented decomposition of the operating system structures explicitly reflects the independence (or lack thereof) in the user access patterns.

2.2 Clustered Objects

Although the object-oriented structure of K42 can help improve locality by mapping independent resources to independent objects, some components may be widely shared and hence require additional measures to ensure locality and good performance. For these objects, K42 uses distributed implementations that either partition and distribute object data 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 distributed implementations. Each clustered object appears to its clients as a regular object, but is internally implemented as a number of *representative* objects that partition and/or replicate the state of the clustered object across processors. Thus, Clustered Objects are conceptually similar to design patterns such as Facade [Gamma et al. 1995] or the partitioned object models used in Globe [Homburg et al. 1995] and SOS [Shapiro et al. 1989]. However, Clustered Objects in K42 are specifically designed for shared-memory multiprocessors as opposed to loosely-coupled distributed systems, and focus primarily on maximizing SMMP locality. The infrastructure used to support Clustered Objects has been described previously [Appavoo 2005; Appavoo et al. 2002; Gamsa et al. 1999] but is summarized here as background for later discussion.

2.2.1 *Details of Clustered Object Implementation.* Each Clustered Object class defines an interface to which every implementation of the class conforms. We use a C++ pure virtual base class to express a Clustered Object interface.

An implementation of a Clustered Object consists of two portions: a Representative definition and a Root definition, expressed as separate C++ classes. The Representative definition defines the per-processor portion of the Clustered Object. In the case of the performance counter, it would be the definition of one of the subcounters. An instance of a Clustered Object Representative class is called a Rep of the Clustered Object instance. The Representative class implements the interface of the Clustered Object, inheriting from the Clustered Object's interface class. The Root class, on the other hand, defines the global, shared, portion of the Clustered Object. Every instance of a Clustered Object has exactly one instance of its Root class that serves as the internal central anchor or "root" of the instance. Each Rep has a pointer to the Root of the

Clustered Object instance. The methods of a Rep can access the shared data and methods of the Clustered Object via its root pointer.

At runtime, an instance of a given Clustered Object is created by instantiating an instance of the desired Root class.[4] Instantiating the Root establishes a unique *Clustered Object Identifier* (COID, also referred to as a Clustered Object ref) that is used by clients to access the newly created instance. To the client code, a COID appears to be an indirect pointer to an instance of the Rep Class.[5]

Tables and protocols are used to manipulate calls on a COID to achieve the runtime features of Clustered Objects. There is a single shared table of Root pointers called the Global Translation Table and a set of Rep pointer tables, one per processor, called Local Translation Tables. The virtual memory map for each processor is set up so that that processor's Local Translation Table appears at a fixed address "`vbase`" on that processor.[6] The Local Translation Table is mapped at the same virtual address on each processor, but its entries can have different values on different processors. Hence, the entries of the Local Translation Tables are processor-specific despite occupying a single range of fixed addresses.

When a Clustered Object is allocated, its root is instantiated and a reference to it is installed into a free entry in the Global Translation Table. (Lists of free entries are managed on a per-processor basis, so that in the common case an entry can be allocated without communication or synchronization.) The address of the corresponding entry in the Local Translation Table is the COID for the new Clustered Object. The sizes of the global and local tables are kept the same, and simple pointer arithmetic can be used to convert between global and local table pointers. Figure 3 illustrates a Clustered Object instance and the translation tables.

Reps of a Clustered Object are created lazily. They are not allocated or installed into the Local Translation Tables when the Clustered Object is instantiated. Instead, empty entries of the Local Translation Table are initialized to refer to a special handcrafted object called the Default Object. The first time a Clustered Object is accessed on a processor (or an attempt is made to access a nonexistent Clustered Object), the same global Default Object is invoked. The Default Object leverages the fact that every call to a Clustered Object goes through a virtual function table.[7] The Default Object overloads the method pointers in its virtual function table to point to a single trampoline[8] method. The trampoline code saves the current register state on the stack, looks up the

[4]The client code is not actually aware of this fact. Rather, a static *Create* method of the Rep class is used to allocate the root. Because we do not have direct language support, this is a programmer-enforced protocol.

[5]To provide better code isolation, this fact is hidden from the client code with the macro: `#define DREF(coid) (*(coid))`. For example, if `c` is a variable holding the COID of an instance of a clustered performance counter that has a method `inc`, a call would look like: `DREF(c)->inc()`. Again, as we do not have direct language support for Clustered Objects we rely on programmer discipline to access a Clustered Object instance only via the `DREF()` macro.

[6]In K42, a page table is maintained on a per-processor and per-address space basis, and thus each processor can have its own view of the address space.

[7]Remember that a virtual base class is used to define the interface for a Clustered Object.

[8]Trampoline refers to the redirection of a call from the intended target to an alternate target.

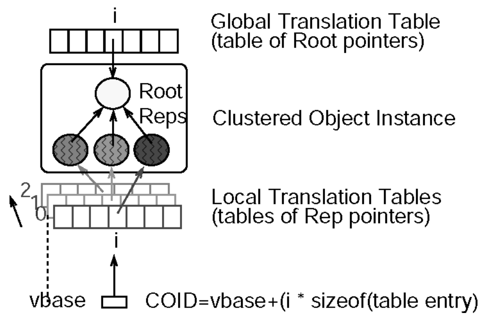

Fig. 3. A Clustered Object instance and translation tables.

Root installed in the Global Translation Table entry corresponding to the COID that was accessed, and invokes a well-known method that all Roots must implement called `handleMiss`. This method is responsible for installing a Rep into the processor's Local Translation Table entry corresponding to the COID that was accessed. The miss handler may instantiate a new Rep or identify a preexisting Rep, in either case storing the Rep's address into the Local Translation Table entry to which the COID points. On return from the `handleMiss` method, the trampoline code restarts the call on the correct method of the newly installed Rep.

The process described above is called a Miss, and its resolution Miss-Handling. Note that after the first Miss on a Clustered Object instance, on a given processor, all subsequent calls on that processor will proceed as standard C++ method invocations, albeit with an extra pointer dereference. Thus, in the common case, methods of the installed Rep will be called directly with no involvement of the Default Object.

The mapping of processors to Reps is controlled by the Root Object. A shared implementation can be achieved with a Root that maintains one Rep and uses it for every processor that accesses the Clustered Object instance. Distributed implementations can be realized with a Root that allocates a new Rep for some number (or cluster) of processors, and complete distribution is achieved by a Root that allocates a new Rep for every accessing processor. There are standard K42 Root classes that support these scenarios.

In K42, all system resources are represented by Clustered Objects so that they can be optimized as the need arises.

2.3 Other Relevant K42 Infrastructure

In addition to K42's object-oriented structure and the clustered object infrastructure, the following three features are also important to K42's scalability:

—**Locality-aware memory allocator**: Using a design similar to that of [Gamsa et al. 1999], our allocator manages per-processor memory pools but also minimizes false sharing by properly padding allocated memory. It also provides for NUMA support, although this does not come into play in the experiments presented here. The memory allocator itself is 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 [Liedtke et al. 1997]. The PPC facility is carefully crafted to ensure that a request from one address space to another (whether to the kernel or to a 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 Krieger et al. [1993] and Gamsa et al. [1999].

—**Automatic object collection**: Object destruction is deferred until all currently running threads have finished.[9] By deferring object destruction this way, any object can be safely accessed, even as the object is 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 basic algorithm of our object destruction facility is described in Gamsa et al. [1999] and the details of the K42 clustered object destruction protocols are described in Appavoo [2005].

Overall, K42 is designed to be demand-driven on a per-processor basis; that is, when a user process makes a request to the system, objects necessary to represent the resources for the request are created on demand on the processor on which the request was made. Future requests by the user process, which require the same resources, will access the same objects on the same processor unless the process is migrated. When the user process terminates, the objects created will be destroyed on the same processor. The K42 scheduler avoids migration in order to encourage affinity with respect to process requests and the objects created to service them. To illustrate this property we recorded the sets of processors on which objects are created, accessed, and destroyed for a 24-way run of the multiuser benchmark discussed in Section 5. Of the 543,029 objects that were created and destroyed during the benchmark, 99.4% were created, accessed, and destroyed on the same processor.

3. PRINCIPLES

Based on our experience developing scalable data structures, we have concluded that:

—Locality must be a central design focus in the effort to achieve robust scalable performance.

—A consistent model for designing system components permits and encourages a pervasive use of locality.

[9]K42 is preemptable and has been designed for the general use of RCU techniques [McKenney and Slingwine 1998] by ensuring that all system requests are handled on short-lived system threads.

—The introduction of distribution within object-oriented encapsulation boundaries eases the burden of constructing scalable systems software.

In this section, we discuss these points in detail.

3.1 Focus on Locality

When constructing a system for which scalability is a requirement, it is worth explicitly considering the implications of concurrency on the software design and implementation. Unless there is a fundamental change in SMMP hardware, data sharing will continue to have an overhead that grows with the number of processors accessing the shared data. As such, the approaches to addressing concurrency should be analyzed with respect to shared data access. There are two basic approaches to constructing concurrent systems:

—Synchronization-driven: Start with a functional uniprocessor system and add synchronization to permit safe, concurrent execution; improve concurrency by incrementally replacing coarse-grain with finer-grain synchronization. Inherently, this approach uses pervasive sharing: the locks and data they protect are typically candidates for access on all processors. Usually no attempt to structurally or algorithmically limit the set of processors that access a lock or data item is made explicitly.

—Locality-driven: Design and implement the system around techniques that permit fine-grain control of sharing. Specifically, distribution and replication are used to improve locality by eliminating or reducing sharing on critical system code paths. This approach does not rely on fine-grain synchronization on performance-sensitive paths.

Most systems have taken the first, synchronization-driven, approach. Given the nature of this approach, it leads to data structures that are accessed by many processors. Good concurrency does not necessarily imply locality, nor does it ensure good scalability given the nature of modern SMMPs. We contend that the widely shared structures that the synchronization-driven approach favors are inflexible with respect to achieving good scalability. All operations that access a shared structure will suffer the communication and cache consistency overheads associated with sharing. On the other hand, using a localization approach allows the system developer to adjust/tune/trade off when and where the overheads of communication will be suffered. Critical operations can be tuned to eliminate shared data accesses.

As an example, consider the problem of maintaining the mapping from file offsets to the physical pages that cache portions of the file. One possibility is a simple hash table. Such a hash table can be constructed to require very small critical sections and thus have good concurrency. However, every access to the hash table would result in shared data access. An alternative that reduces the shared data access is to maintain a master hash table and replicate the data to local per-processor hash tables. As the local hash tables become populated, lookups require only local memory access, and sharing is avoided.

There are, of course, limitations to localizing implementations: one can not eliminate all sharing; one must be selective about which paths will be optimized.

This approach inherently demands fine-grain trade-offs. Optimizations often involve extra complexity and space usage, and optimizing some operations may decrease the efficiency of others. Returning to our hash table example, the proposed distributed implementation improves the locality of the lookup operation at some cost in space and complexity. It also greatly increases the cost of the remove operation, because now a mapping must be removed from all the local hash tables. Optimization decisions must always take into account the frequencies with which different paths are executed.

The locality-driven approach is not unique or new, and many systems have utilized distribution to solve particular problems or have applied it in the large for the sake of addressing extreme disparity in communication/computation latency (in particular, for distributed systems). We decided to take this approach in addressing scalability on SMMPs, because even on machines that are considered to have good computation/communication ratios, the costs of sharing can dominate can dominate the performance of many operations.

3.2 Utilize a Model

In our effort to address performance problems by improving locality, we have found that it was key to first develop a model for distributed implementations. In K42 we adopted the Clustered Object model, using it as the basis on which to approach each new implementation. The Clustered Object model gave us a concrete set of abstractions with which to reason and design solutions. Solutions for each performance problem could be approached within the context of the model, with the goal of providing localized processing on performance-critical paths. A key feature of the model is its ability to motivate and guide the developer in achieving good locality. Our current Clustered Object model has been developed from experience and has specifically been designed to be simple.

The Clustered Object model helps encourage the developer to address questions. Specifically, it forces the developer to answer the following:

—What state should be placed in the Clustered Object Representatives and hence be local?

—What state should be shared and placed in the Clustered Object Root?

—How should the Representatives be created and initialized? At what point should new Representatives participate in the global operations of the object?

—How should local operations and global operations synchronize?

—How should the Representatives be organized and accessed?

For example, consider constructing the in-kernel data structures that represent a process. As one declares the data fields that are associated with a process and the operations in which they are used, the Clustered Object model forces the developer to actively decide if and how to utilize distribution. If the developer decides not to distribute any data fields, then she uses a single representative, containing all data members, for all processors. If, however, it becomes clear that servicing page faults incurs contention on the list of valid memory mappings for the process, the Clustered Object model gives the developer a basis for distributing the list of valid memory mappings. Introducing a per-processor local copy of the list of valid memory mappings is naturally achieved by using multiple Representatives. The remainder of the data fields are placed in the Root. Additionally, the Clustered Object model forces the developer to consider how the process data structures should scale as new processors access the process object. The developer must define how new Representatives are created when the process object is accessed for the first time on a processor. Doing so, she must also decide how the new Representative's copy of the valid mappings will be initialized and how and when global operations such as removing a mapping will coordinate with the new representative.

3.3 Encapsulate Distribution

Having adopted object-oriented decomposition and a means for integrating distribution, we constructed our system with a focus on improving locality. Supporting distribution within the encapsulation boundaries of an object-oriented language preserved a number of object-oriented advantages.

Critical to operating systems development is the need to allow for incremental development and optimization. By encapsulating the use of distribution within a component's internals, we have been able to iteratively introduce distribution. For example, we were able to construct the virtual memory management subsystem as a set of object implementations which utilized little or no distribution. As experience was gained with each object, we iteratively changed the implementation of individual objects to introduce more aggressive distribution on a per-implementation basis without breaking interoperability. A specific example is the Global Page Manager object that manages the system-wide pool of physical page frames. Over time we progressively localized more and more of its functionality by locating its data members into its Representatives and rewriting its methods. Each version we produced was functionally equivalent to its predecessors and was able to participate in a functioning system without impacting any other object implementations.

Encapsulating the distribution preserves the ability to leverage polymorphism. It allows us to selectively instantiate implementations with differing distribution trade-offs for the same resource type. For example, a page cache for a dynamic region, such as the heap for a single-threaded application, is best served by a data structure that is optimized for the miss case and utilizes a centralized structure. The page cache for a long-lived, multiply accessed set of pages, such as those of an executable or data file, is best served using a data structure that is optimized for the hit case and utilizes a distributed structure.

Finally, the encapsulation of the distribution preserves an object-oriented functional interface model for the software components—the only access to an object is through its method interface. We have leveraged the uniform functional interface and associated calling convention to construct specialized objects that can be interposed dynamically in front of any object regardless of its type. An instance of such an interposer can intercept calls to the particular object it fronts and can implement arbitrary call handling including: invocation of arbitrary functions, manipulation of arguments, redirection of the call, passing of the call on to the original target, and manipulation of return values. We have used interposer objects to implement:

—**Lazy distributed component construction** On first method invocation of an object on a processor, an object-specific resource allocation method is called to locate or construct a representative of the given object for the processor. Successive calls to the object go directly to the representative with no interposition.

—**Distributed resource reclamation** When an object is destroyed, a special object is interposed in front of the object that is being destroyed. The special object returns a standard error code back to all callers without ever accessing the original object. When the destruction system has reclaimed all the resources associated with the object and has determined that there are no threads that have a reference to the object, the special object is removed.

—**Hot-swapping of distributed objects** We have implemented a methodology for dynamically swapping compatible object instances, including instances which have a distributed representative structure in an efficient multiprocessor fashion [Hui et al. 2001; Soules et al. 2003].[10]

4. MEMORY MANAGEMENT CASE STUDY

Virtual memory management (VMM) is one of the core services that a general purpose operating system provides and is typically both complex in function and critical to performance. Complexity is primarily due to (i) the diversity of features and protocols that the virtual memory services must provide, and (ii) the highly asynchronous and concurrent nature of VMM requests. The demand for performance is primarily on one VMM path—the resident page fault path. Generally, each executing program must establish access to the memory pages of its address space via page faults. "Resident" page faults are those that can be resolved without involving disks or other slow I/O devices, either because the content of the required pages can be computed (e.g., zero-filled or copied from other pages) or because the content is already resident in memory and can simply be mapped into the requester's address space.

The optimization of the K42 VMM service, to improve its scalability, has served as a case study for the application of distributed data structures to a core operating system service. In this section, we review how distributed data structures encapsulated in the Clustered Objects of the K42 VMM are used to achieve the optimizations.

Figure 4 depicts the K42 Kernel VMM objects used to represent and manage the K42 virtual memory abstractions. Specifically, the figure illustrates the instances of objects and their interconnections that represent an address space with two independent files (File1 and File2) mapped in, as shown abstractly at the top of the diagram above the dashed line. The associated network of kernel objects is shown below the dashed line. Some of these objects were discussed briefly in Section 2.1.

[10]A methodology for doing general dynamic system update built on top of hot-swapping is being explored by Baumann et al. [2005].

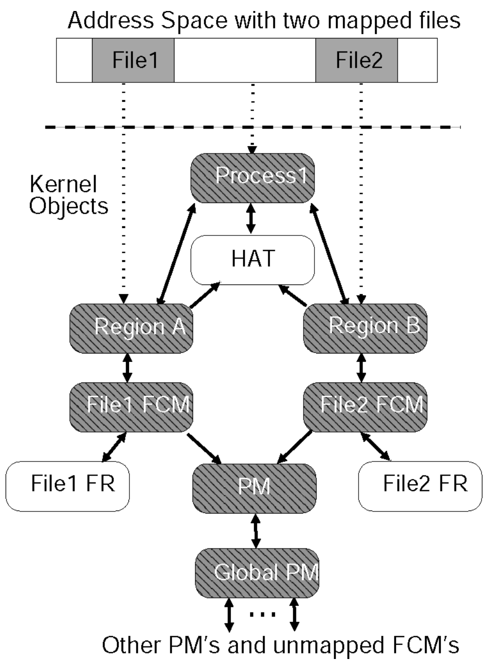

Fig. 4. VMM object network representing an address space with two mapped files.

We have reimplemented each of the shaded objects (Process, Region, File Cache Manager (*FCM*), Page Manager (*PM*) and Global Page Manager (*Global PM*)) in a distributed fashion and each is discussed in subsequent subsections. The other two objects, HATs, and FRs, are discussed in the following two paragraphs.

HAT. The Hardware Address Translation (HAT) object is responsible for implementing the functions required for manipulating the hardware memory management units and is used to add a page mapping to the hardware-maintained translation services. In K42, the HAT object maintains page tables on a per-processor basis. Hence, the default HAT implementation is distributed by definition and naturally does not suffer contention.

FR. The File Representative (FR) objects are in-kernel representations of files. An FR typically communicates with a specific file system server to perform the file IO. The only situations where FRs are heavily utilized are under circumstances where the system is paging. Those scenarios may lead to more substantial sharing of the FRs, in which case similar techniques to those we describe would need to be applied. To date, we have not run benchmarks that have placed significant paging burdens on the system to observe this potential effect.

In order to gain insight into the performance implications of this implementation, let us consider a microbenchmark that we will call **memclone**.[11]

[11]This benchmark was sent to us by one of IBM's Linux collaborators. The benchmark was developed to demonstrate scalability problems in Linux.

Memclone is designed to mimic the page-fault-inducing behavior of a typical scientific application's initialization phase. For each processor, it allocates a large array and a thread that sequentially touches each array page. To understand the implications of K42's VMM structure on the performance of such a benchmark, we will briefly trace the behavior of K42's VMM objects during execution of the program.

When each thread allocates its large array (on the order of 100 Megabytes) the standard Linux C library implementation of `malloc` will result in a new memory mapping being created for each array. In K42, this mapping will result in a new Region and File Cache Manager (FCM) being created for each array. The FCM records and manages the physical pages that are backing the resident data pages for the mapping. The FCM is attached to an FR which simply directs the FCM to fill newly accessed page frames with zeros.

Thus, running the benchmark on n processors will result in the Process object for the benchmark having n Regions, one per array, added to its list of regions. As each thread of the benchmark sequentially accesses the virtual memory pages of its array, page fault exceptions occur on the processor on which the thread is executing.

K42's exception handler directs each fault to the Process object associated with the benchmark by invoking a method of the Process object. The Process object will search its list of regions to find the Region responsible for the address of the page that suffered the fault and then invoke a method of the Region. The Region will translate the faulting address into a file offset and then invoke a method of the FCM it is mapping. The FCM will search its data structures to determine whether the associated file page is already present in memory. In the case of the page faults induced by the benchmark, simulating initialization of the data arrays, all faults will be for pages that are not present. As such, the FCM will not find an associated page frame and will allocate a new page frame by invoking a method of the Page Manager (PM) to which it is connected. The PM will then invoke a method of the Global PM to allocate the page frame. To complete the fault, the FCM initializes the new page frame with zeros, and then maps the page into the address via the HAT object passed to it from the Process.

In the following subsections, we review design alternatives for each of the major objects and present the results of measurements to identify the effects of data distribution and replication.

4.1 Process Object

The Process object represents a running process and all per-process operations are directed to 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 of Regions in order to direct the fault to the appropriate Region object. The left-hand side of Figure 5 illustrates the default nondistributed implementation of the Process object. A single linked list with an associated lock is used to maintain the Region List.

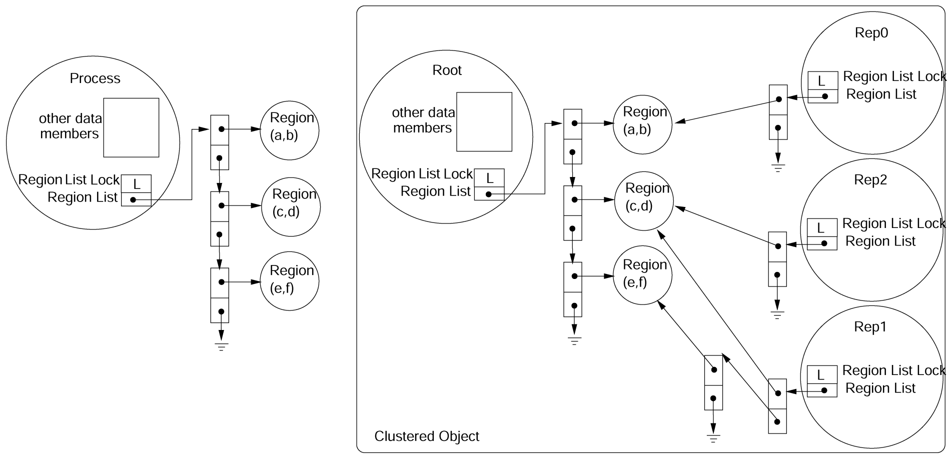

Fig. 5. Nondistributed and distributed Process objects.

To ensure correctness in the face of concurrent access, this lock is acquired on traversals and modifications of the list.

In the nondistributed implementation, the list and its associated lock can become a bottleneck in the face of concurrent faults. As the number of concurrent threads is increased, the likelihood of the lock being held grows, which can result in dramatic performance drop-offs as threads stall and queue on the lock. Even if the lock and data structures are not concurrently accessed, the read-write sharing of the cache line holding the lock and potential remote memory accesses for the region list elements can add significant overhead.

The distributed variant of the Process object is designed to cache the region list elements in a per-processor Representative (see right-hand side of Figure 5). A master list, identical to the list maintained in the nondistributed 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 is 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 that has suffered a previous fault on a processor) will only access memory local to the processor and not require any inter-processor communication or synchronization.

Using distribution is not the only alternative. Replacing the coarse-grain lock and linked list with a fine-grain locked list or lock-free list would likely yield similar benefit. Our goal, however, is not to consider all possible implementations of a concurrent linked list but rather to establish that locality-based solutions are viable.

Figure 6 displays the performance of memclone (described earlier) running on K42 with the nondistributed version and the distributed version of the Process object. The experiments were run on an S85 Enterprise Server IBM

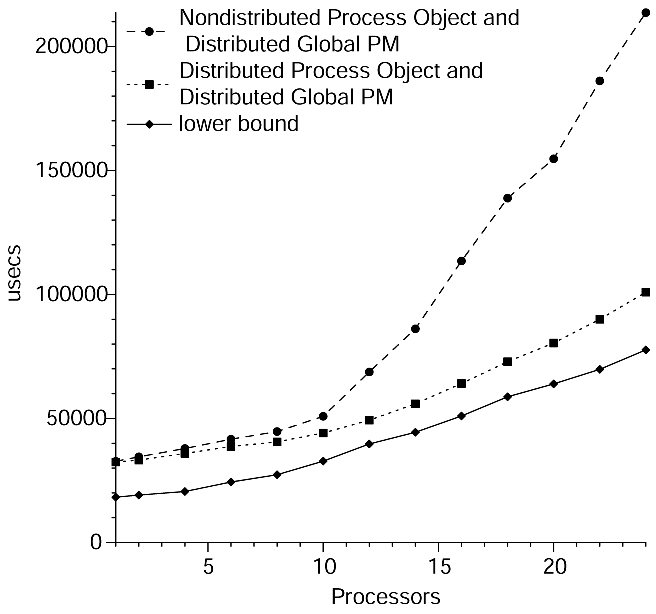

Fig. 6. Graph of average thread execution time for memclone running on K42. One thread is executed per processor. Each thread touches 2500 pages of an independent memory allocation. Each page touch results in a zero-fill page fault in the OS. The performance of using non-distributed and distributed Process objects is shown. In both cases the distributed Global PM object is used. The lower-bound for concurrent zero-filling of memory on the S85 is included for reference.

RS/6000 PowerPC bus-based cache-coherent multiprocessor with 24 600 MHz RS64-IV processors and 16GB of main memory.

In these experiments, each thread of memclone touches 2500 pages of memory and one thread is executed per processor. The average execution time is measured and plotted in the graph. Ideal scalability would correspond to a horizontal line on the graph. We see that beyond 10 processors, the performance of the nondistributed version degrades rapidly, whereas the distributed Process object achieves significantly better scalability.

Scalability is not perfect because of subtle inherent hardware bottlenecks. To isolate this effect, we constructed an in-kernel, zero-fill test that measures the time it takes each processor to map and zero-fill an independent set of 2500 pages. On each processor we ran a kernel thread that executes a *memset0* routine over a processor-specific 2500-page test memory region. The *memset0* routine is architecture specific, and the PowerPC version used on our test hardware utilizes the PowerPC data cache block zero (dcbz) instruction to efficiently zero an entire cache line at a time. The kernel page fault handler was modified to use a fixed calculation to map virtual to physical pages for the test memory region. As such, page faults to the test region do not require any data structure access.

The bottom line in Figure 6 plots the performance of the zero-fill test. We see that there is an intrinsic nonscalable cost for a parallel zero-filling workload on our hardware. Since the core function that memclone exercises is the parallel zero-filling of newly allocated user memory, the lowest line in Figure 6 is our lower bound. The key difference for the memclone experiment compared to the kernel zero-filling experiment is that each page fault induced by memclone is a user-level page fault that requires kernel data structures to resolve. As such, when evaluating memclone performance we do not expect to achieve the lower bound but rather deem that scalable performance has been achieved if we are adding a constant overhead that is independent of the number of processors.

The distributed object has greater costs associated with region attachment and removal than the nondistributed implementation. For a region that is not accessed, the distributed version adds approximately 1.2 microseconds to the time of 20.19 for mapping and 0.85 microseconds to the time of 11.12 microseconds for unmapping over the nondistributed version. In the worst case, to unmap a region that has been accessed on all 24 processors, the distributed version adds 73.1 microseconds to the time of 153 microseconds for the nondistributed version. In the case of a multithreaded process, the overhead of the distributed implementation for region attachment and removal has not proven to be problematic. For 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. In Soules et al. [2003] we describe techniques for dynamically swapping one instance of a Clustered Object for another and believe that this approach could be used if the overheads of the distributed implementation become prohibitive.

4.2 Global Page Manager Object

A hierarchy of Page Managers (PM) is used in K42. There is a PM associated with each process, and a Global PM at the root that is responsible for physical page frame management across all address spaces in the system.[12] The PM associated with each process manages and tracks the physical page frames associated with the process and is responsible for satisfying requests for page frames as well as for reclaiming pages when requested to do so. It also tracks the list of computational FCMs (FCMs that manage pages of anonymous mappings) created by the process. Currently, the per-processor PM simply redirects memory allocation requests to the Global PM.[13]

FCMs for named files opened by processes are attached to the Global Page Manager. The Global Page Manager implements reclamation by requesting pages back from the FCMs attached to it and from the Page Managers below it.

The left-hand side of Figure 7 illustrates the simple, nondistributed 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 FCMs and PMs. All three data structures are protected by a single, shared lock. On allocation and deallocation requests, the lock is acquired and the free list manipulated. Similarly, when a PM or FCM is attached or removed, the lock is acquired and the appropriate hash table updated. Reclamation is implemented

[12]K42 employs a working set page management strategy.

[13]The main purpose of the PM associated with each process is to provide a degree of freedom for future exploration of more complex VMM implementations.

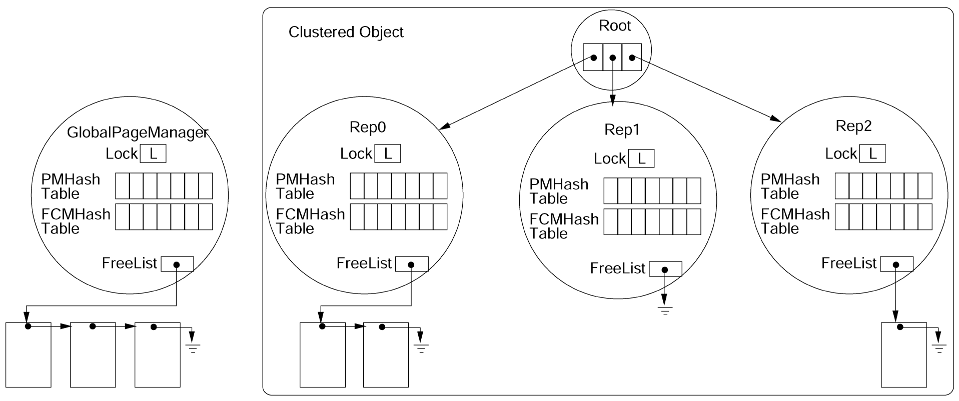

Fig. 7. Nondistributed and distributed Global Page Managers.

as locked iterations over the FCMs and PMs in the hash tables with each FCM and PM instructed to give back pages during the reclamation iterations.

As the system matured, we progressively distributed the Global Page Manager object in order to alleviate the contention observed on the single lock. The most recent distributed implementation is illustrated on the right-hand side of Figure 7. The first change was to introduce multiple Representatives and maintain the free lists on a per-processor basis. The next change was to distribute the FCM Hash Table on a per-processor basis by placing a separate FCM Hash Table into each Representative and efficiently mapping each FCM to a particular Representative.

In the distributed version, page allocations and deallocations are done on a per-processor basis by consulting only the per-Representative free list in the common case, which improves scalability. The free lists per-Rep are limited to a maximum size, and an overflow list is maintained at the root to allow free frames to be moved between processors.

The mapping of FCMs and PMs to an appropriate Global PM Representative is illustrative of the distributed techniques exploited to avoid contention. The underlying clustered object infrastructure provides mechanisms to cheaply identify the processor where an object was created. The allocating processor for an FCM/PM is treated as its home processor, and the representative at that home processor keeps track of that object. State is naturally distributed across the representatives, with natural locality if the processor where an object is created is the same processor where it is typically accessed.

In Figure 8, we plot the performance of an instrumented version of memclone, measuring the average execution time of the threads using the various combinations of Process and Global PM implementations. Each thread, again, touches 2500 pages of memory, and one thread is executed on each processor. Ideal scalability would be a horizontal line on the graph. Focusing on the two curves labeled *Distributed Process and Nondistributed Global PM Objects* and *Distributed Process and Distributed Global PM Objects* we can consider the performance impact of the Global PM implementations. The nondistributed version results in a sharp decrease in performance beyond 8 processors. The

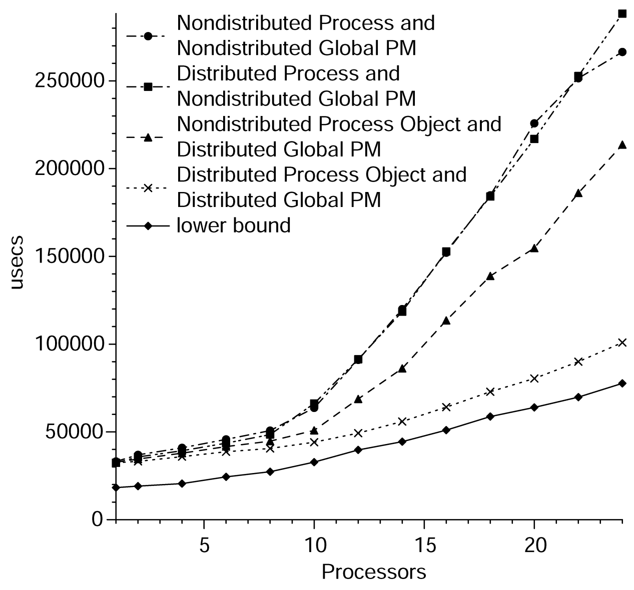

Fig. 8. Similar to Figure 6. Graph of average thread execution time for memclone running on K42.

distributed version yields scalable performance that mirrors the limits of the underlying hardware.

When we consider all curves of Figure 8, it is clear that using only one of the distributed implementations is not sufficient. The use of distributed implementations of both the Process objects and the Global PM object are required in order to obtain a significant improvement in scalability. This phenomenon is indicative of our general multiprocessor optimization experience, where optimizations in isolation often affect performance in nonintuitive ways; that is, the importance of one optimization over another is not obvious. A key benefit of the Clustered Object approach is that every object in the system is eligible for incremental optimization, not just the components or code paths that the developer might have identified a priori.

4.3 Region

The Region object is responsible for representing the mapping of a portion of a file to a portion of a process's address space. It serves two purposes. First, it translates the address of a faulting page to the appropriate offset into the file that is being mapped at that portion of the address space. Second, it provides synchronization between faults on that region and requests to create, resize, and destroy regions.

For synchronization, the original version of the Region object uses a form of reference counting. A single request counter is used to record all relevant requests (method invocations). At the start of a request, an `enter` operation is performed and on exit a `leave` operation is performed on the counter. Depending on the state of the Region the `enter` operation will: (*i*) atomically increment the counter, or (*ii*) fail, indicating that the request should be rejected, or (*iii*)

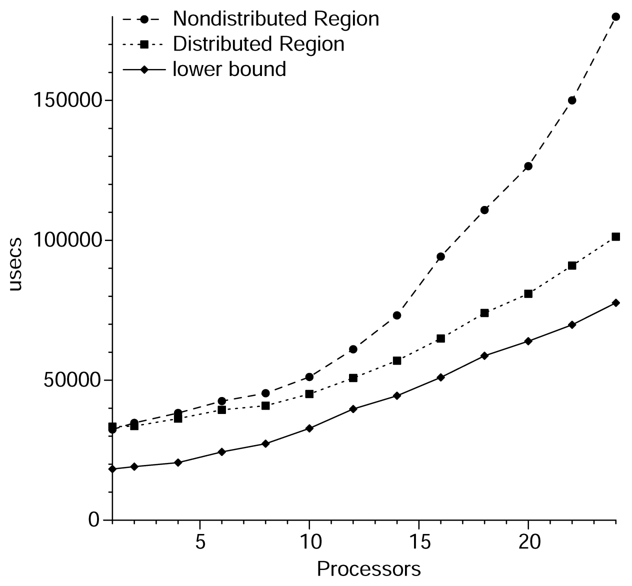

Fig. 9. Graph of average thread execution time for a modified memclone. One thread is executed per processor. Each thread touches 2500 independent pages of a **single** large common memory allocation. Each page touch results in a zero-fill page fault in the OS. The graph plots the performance when using a nondistributed versus a distributed Region object to back the single common memory allocation. The lower bound performance for concurrent zero-filling is included for reference.

internally block the request. The behavior of the request counter is controlled by a control interface. Ignoring the details, the request counter is fundamentally a single shared integer variable that is atomically incremented, decremented and tested in all relevant methods of the Region object, including the `handleFault` method that is on the resident page fault path.

Figure 9 displays a graph of the performance obtained when running a modified version of memclone on K42 using the nondistributed versus distributed implementation of the Region object. In the modified version of memclone, instead of allocating their own independent arrays, the threads access disjoint 2500-page segments of a single large allocated array. The zero-fill page faults that the threads induce will therefore be serviced not only by a common Process object and the Global PM but also by a common Region and the FCM representing the single memory allocation. For the experiments represented in the figure, all objects other than the Region backing the allocation are using distributed implementations so that we can focus on the impact of the Region in isolation.

As Figure 9 shows, there is a performance degradation with the original version of the region. This performance degradation occurs even though there are no locks in the Region for the page fault path and the critical section, the atomic increment of a shared reference count, is small compared to the overall cost of a page fault (which in the case of memclone includes zero filling the page frames). This example demonstrates the importance of removing shared accesses on critical paths even if an implementation with shared accesses has good concurrency.

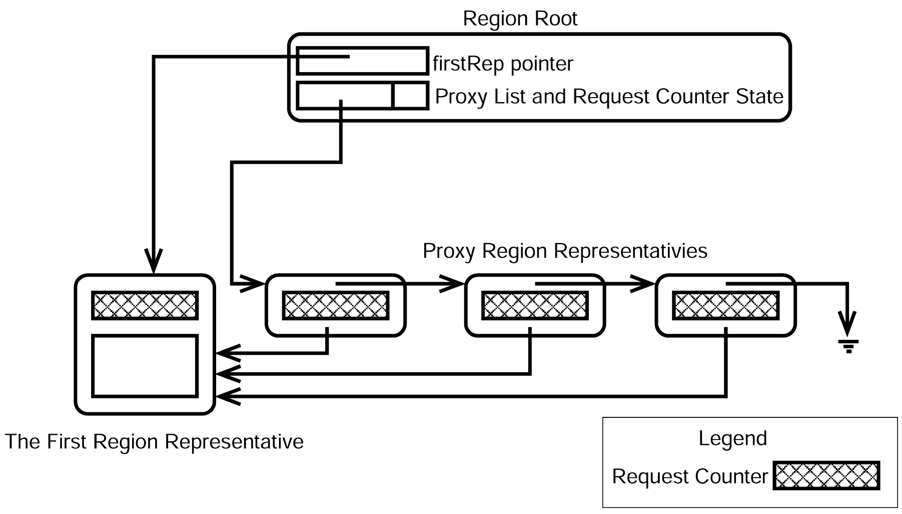
Region Root
firstRep pointer
Proxy List and Request Counter State
Proxy Region Representativies
The First Region Representative
Legend
Request Counter

Fig. 10. Distributed Region Clustered Object. The first Region Representative encapsulates the core functionality and all other Representatives are proxies that contain only a request counter. Most calls to the proxy are simply forwarded to the firstRep; however, the handleFault method manipulates the request counter of the proxy and then forwards its call to a method of the firstRep that implements the main logic of the call. Thus, with respect to the fault handling path the only read-write variable is manipulated locally. The root maintains the list of proxies as well as the state of the request counters globally in a single variable that can be updated atomically.

The distributed version that we constructed addresses the problem by using a per-processor reference count that is independently incremented on the page fault path (see Figure 10). It creates a single Representative (the *firstRep*) when the distributed Region object is created. The Root for the distributed version is constructed to create specialized Representatives, which we refer to as proxy Representatives, for any additional processor that may access the Region instance. For all methods other than the crucial page fault method, the proxy Representatives invoke the centralized method on the *firstRep*. In the case of the `handleFault` method, which is the only method that increments the reference counter, the proxy Representative version does an increment on its local request counter and then invokes the body of the standard `handleFault` method on the *firstRep*.

The majority of the complexity and new functionality associated with the distributed Region is factored into the root class. The root object's methods are constructed to ensure correct behavior in the face of dynamic additions of Representatives. As is standard for distributed Clustered Objects, new accesses to the Region on other processors can require the creation of new Representatives via a standard instantiation protocol. The protocol requires that a developer specify a `createRep` method in the root of the Clustered Object that creates and initializes a new Representative for the processor that accessed the object. In the case of the distributed Region's root, this method creates a proxy Representative and then initializes the state of the request counter associated with the new Representative.

As can be seen in Figure 9, with this experiment the distributed implementation of Region results in scalability that tracks the capability of the hardware.

4.4 File Cache Managers

Each region of an address space is attached to an instance of a File Cache Manager (FCM), which caches the pages for the region.[14] An FCM is responsible for all aspects of resident page management for the file it caches. On a page fault, 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 unmapping,
(2) asynchronous I/O between the faulting process and the file system,
(3) timely and efficient page reclamation, and
(4) maintenance of UNIX fork relationships[15] for anonymous files.

The standard, nondistributed FCM uses a single lock to reduce the complexity of its internal implementation. When a file is accessed by a single thread, the lock and centralized data structures do not pose a problem, but when many threads access a file concurrently, the shared lock and data structures degrade page fault performance.

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

As illustrated in Figure 11, a reusable distributed hash table (DHash) was designed with two basic components, MasterDHashTable and LocalDHashTables, designed to be embedded into a Clustered Object's root and representatives, respectively. DHash has a number of interesting features:

(1) LocalDHashTables and MasterDHashTables 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;

[14]An FCM may be backed by a named file or swap space (for anonymous regions).
[15]When a process executes a UNIX fork we must ensure that the anonymous memory mappings, such as the heap, preserve the hierarchical relationships between parent and children as implied by UNIX fork semantics.

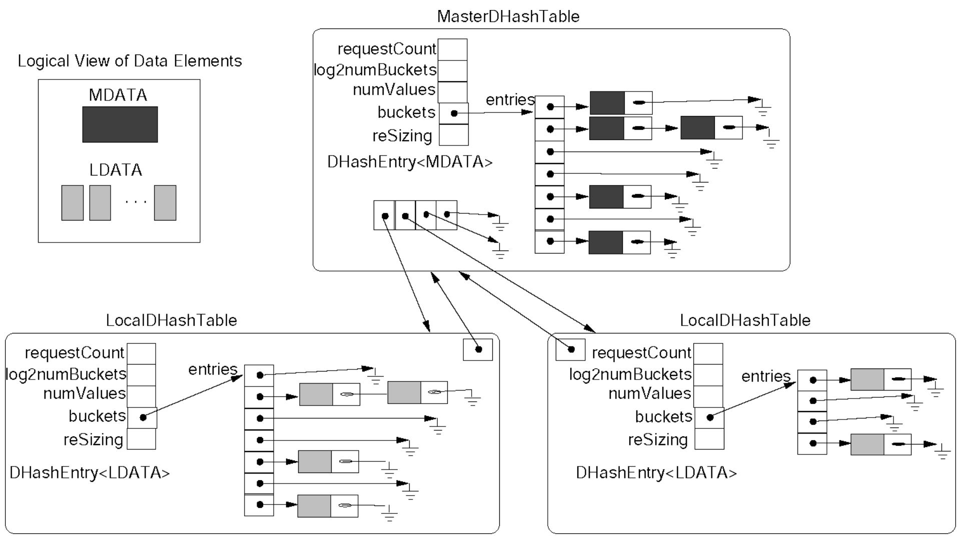

Fig. 11. Reusable distributed hash table Clustered Object components.

(4) fine-grain locking is used—common accesses require an increment of a reference count and the locking of a single target data element;

(5) data elements with both shared and distributed constituents are supported; and

(6) scatter and gather operations are supported for distributed data elements.

Our distributed hash table was designed and implemented with shared-memory multiprocessor synchronization techniques, operating system requirements, and SMMP locality in mind so that it can be reused in the construction of other Clustered Objects. As such, its constituent subparts are designed to cooperate within the context of a containing Clustered Object. The hash was been implemented to permit the local portions to be dynamically created when a representative is created, and it was designed to provide a hot lookup path that does not require remote memory accesses or synchronization. Section 6.2.4 discusses related work with respect to hash tables.

Although a complete description of the internals and interface of DHash are beyond the space limitations of this paper, a brief description of the two key methods follows:

`findOrAllocateAndLock(key)`: query the hash table for the data item associated with `key`. If it is found, the local copy is returned. Otherwise, a newly allocated local data element is returned. In both cases the local data element is locked on return. It is the caller's responsibility to unlock the item. Internally, DHash ensures race-free allocation by coordinating around the Master-DHashTable, ensuring that only one miss induces an allocation of an entry associated with the `key`.

`emptyData(data)`: logically deletes a data element from the hash tables so that future lookups will fail to find it. Internally, the method ensures that the operation occurs atomically across any replicas of the data element in the various

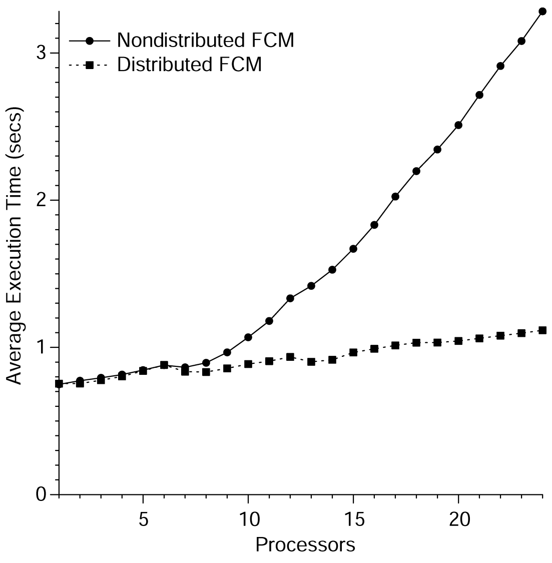

Fig. 12. Graph of distributed FCM and nondistributed FCM performance. One instance of grep is executed on each processor. Each grep searches a common 111MB file for a common nonmatching search pattern. The graph shows the average grep completion time.

hash tables. It also ensures that the data structures storing the data element are destroyed or reused only when it is actually safe to do so, that is, no threads are actively inspecting the associated memory.

The first demands for a distributed FCM came from concurrent access to a shared file from a multiuser workload. This scenario motivated us to develop a distributed FCM based on Dhash, in which the performance-critical path was the remapping of the resident pages of a long-lived file. An FCM implementation using the DHash constituent in a straightforward manner is sufficient in this case, as the majority of page faults will result in accesses only to the local DHash table in order to remap pages which have already been mapped on the processor by a previous execution. This scenario can be explored with a microbenchmark, in which instances of the UNIX “grep” utility are run on each processor, all searching a common file. The programs will induce concurrent page faults to the FCM that caches the pages of the executable, as well as concurrent faults to the data file being searched. Figure 12 illustrates the performance of such a microbenchmark using the nondistributed and the distributed version of the FCM.

When we consider the performance of the modified version of memclone (Figure 13) with the distributed FCM (line labeled Distributed DHash FCM), however, we observe poor scalability. The distributed FCM was designed to optimize the resident page fault path in which the majority of faults would be to pages that already exist in the FCM and have been previously accessed on the processor. The memclone benchmark does not have this behavior. Most of its page faults are to pages that do not yet exist. The local lookups fail and the new zero-filled pages have to be added to the DHash master table. The distributed

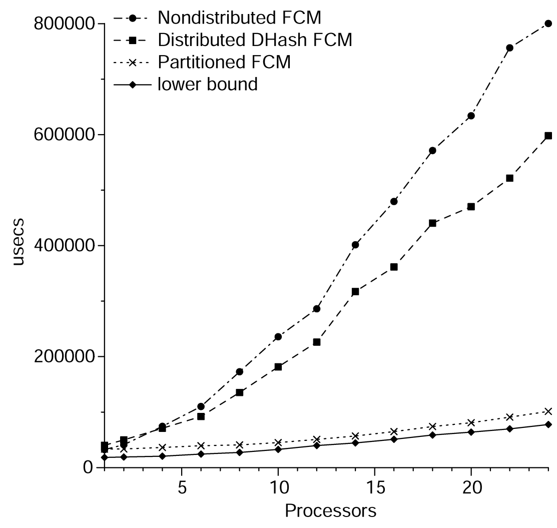

Fig. 13. Similar to Figure 9. Graph of average thread execution time for a modified memclone, as described in the text, running on K42. The graph plots the performance of using a nondistributed FCM versus a distributed FCM, using DHash versus a partitioned FCM, to back the single common memory allocation.

version acts like a central hash table with fine-grain locking, with the additional overhead of creating replicas whose cost is never amortized over multiple accesses. These costs are justified only if there are additional faults to the pages that can be satisfied from the replicas. As noted above, this amortization occurs when a file is reused on a processor, as might be the case for a frequently used executable. It is also possible that a better balance between creation cost and reuse amortization can be achieved by assigning an FCM Rep, not to every processor, but to small subsets of processors. All the processors in a subset would benefit from the local replica created by the first processor to access a page. The Clustered Object infrastructure provides the flexibility needed to explore this type of trade-off. We think an intermediate level of distribution may be appropriate for the randomly accessed heap of a long-lived system server (or server application such as a database), but this exploration is left for future work.

To address the zero-fill scenario presented by memclone, an FCM with a partitioned structure was constructed by reusing parts of the DHash constituent. This version partitions the set of pages across a set of representatives but does not distribute the page descriptors. A fixed number (m) of MasterDHashTables are used to manage the page descriptors for a fixed disjoint set of pages. A new LocalDHashTable class is used to simply redirect invocations to one of the m MasterDHashTables. All representatives have an instance of the LocalDHashTable but only the first m representatives have a master hash table. The policy for page assignment is determined by a specified shift value that is applied to the page offset. The result is then mapped to one of the m master tables via a modulo calculation. A prototype of the partitioned FCM was constructed and its performance is also shown in Figure 13 (dashed line) with much improved scalability.

4.5 Findings

In summary, the K42 infrastructure (i.e., the object-oriented structure and Clustered Objects introduced in Section 2), and the combination of the components described in this section, result in a page fault path that is fully scalable; that is, with our design of the page fault path, two threads of the same process can fault on the same (cached) page of the same file without causing the kernel to access any common memory or acquire any common locks. We would like to stress again that we are not making the argument that designing system software using our methodology and framework necessarily leads to solutions that are better than any alternative solution, but rather that it leads to software that can scale using a reasonably straightforward and manageable software engineering process. Much of the prior work in improving data structure concurrency (such as Gao's work on a lock-free hash table [Gao et al. 2005] and McKenney et al.'s work on *Read-Copy-Update* [McKenney et al. 2001; Torvalds 2002] is critically important, but does not address the issue of locality. Our case study on the Region object in particular shows that just a single shared counter can cause serious performance degradation (and that the motivating example of Section 1.1 is not as contrived as it may seem).

The case studies and accompanying experiments presented in the previous subsections allow us to validate the principles set out in Section 3.

First, we find that it is possible to design software that is reasonably scalable by focusing primarily on locality instead of concurrency. We found that the required concurrency tends to fall out naturally when designing for locality. With extreme microbenchmark stress tests we were able to demonstrate performance and scalability that mirrors the limits of the underlying hardware.

Second, the patterns for deriving locality-optimized solutions in the different case studies were similar. We start with a simple implementation and incrementally add improvements as the need arises. In each case, one decides which data to assign to the root object, which data to assign to the representative objects, and how to coordinate amongst the objects to maintain consistency. Because a consistent methodology was used in each case, the solutions have a common structure even though the problems and solutions differ in detail. Overall, we found that the methodology and framework allowed us to develop solutions that met our scalability requirements with tractable complexity. It is our belief that building a fully distributed page fault path (with similar fine-grain locality) into a traditional operating system without a proper structure and accompanying methodology would result in a system of unmanageable complexity.

Third, by leveraging the encapsulation offered by objects in our framework in general and by Clustered Objects specifically, we were able to optimize individual components in isolation. The code of a target (Clustered) object could be modified without changing other parts of the code.

5. THE LARGER CONTEXT

In Section 4, we used the memclone microbenchmark to allow us to isolate the impact of distributed objects in memory management, comparing the performance of the distributed implementation to that of simpler nondistributed implementations in our system. In this section we discuss this work in a larger context, focusing on (i) overall system performance for three workloads, (ii) other characteristics of the system that are synergistic with (or in conflict with) the OO design and (iii) Clustered Objects.

5.1 System Performance

K42 was designed to explore a structure that would permit the construction and optimization of an operating system for large future systems. Given that we would only be able to test on a multiprocessor of limited scale, we focused our efforts on system structure and techniques for promoting locality and progressively optimizing the locality of critical paths in the system. Our aim was not to produce a scalable OS for commercial machines not yet generally available, but rather to determine a structure and techniques that practically could be used to construct scalable systems software for future machines.

The performance issues examined in previous sections were exposed using a micro-benchmark that aggressively exercised one aspect of the system. It is natural to question whether the techniques we described are of any relevance to smaller systems, and whether they are useful for traditional server workloads. To explore these questions we use three OS-intensive benchmarks. All three benchmarks are composed of multiple independent sequential applications accessing independent directories and files. The benchmarks are designed to be simple and to offer a scalable concurrent load to the system. The chosen benchmarks are not aggressively parallel applications, but are instead typical of the workloads handled by large multiuser UNIX servers. The workloads are inherently scalable and target traditional UNIX functionality. Perhaps more importantly, overall performance is not gated by application-level performance; there is no explicit application concurrency to limit scalability. Analyzing the scalability of an operating system with these benchmarks is appropriate, as system-level function is criticial to their performance.

Large-scale parallel applications, including Web and database servers, are clearly important workloads for SMMP systems. K42 includes specialized support for parallel applications, but that aspect of the system is not the focus of this paper. For the purpose of evaluating the scalability of operating system services, parallel applications are not necessarily ideal experimental workloads. Highly tuned applications that scale well may not stress the OS, and the ones that do not may mask scalability problems in the underlying system.

5.1.1 *Details of Experiments.* All the results in this paper were obtained by running K42 or Linux on PowerPC hardware. We used the IBM S85 eServer pSeries 680 RS/6000 bus-based cache-coherent multiprocessor with 16 GB of main memory and 24 RS64-IV processors clocked at 600 MHz. Cache coherence for this machine relies on distributed bus snooping, not a centralized directory. The distributed mechanism places a cache block in any cache into one of four states using the MESI protocol [Papamarcos and Patel 1984]. The L1 cache is split into instruction and data caches, each of size 128 KB, with a line size of 128 bytes, and 2-way set-associative.

The three benchmarks we ran to examine scalability were SPEC SDET, simultaneous Postmark, and simultaneous make, described in the following. To stress operating-system rather than disk performance, each of the experiments was run using a ramdisk file system. We ran identical experiments on K42 and on Linux 2.4.19 as distributed by SuSE (with the O(1) scheduler patch). Linux kernel components (device drivers, network stack, etc.) that are used inside K42 are from the same version of Linux.

The SPEC Software Development Environment Throughput (SDET) benchmark [spec.org 1996] executes, in parallel, a specified number of "user" scripts, each of which sequentially executes a series of common UNIX programs such as `ls`, `nroff`, `gcc`, `grep`, and so on. We modified the benchmark slightly to avoid the programs `ps` and `df`, which K42 does not yet fully support.

To examine scalability we ran one script per processor. The same program binaries (for `bash`, `gcc`, `ls`, etc.) were used on both K42 and Linux. Glibc version 2.2.5 was used for both systems, but the K42 version was modified 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 executed per hour.

Postmark 1.5 [Katcher 1997] was designed to model a combination of electronic mail, netnews, and Web-based commerce transactions. 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. A separate instance of Postmark was created for each processor, with corresponding separate directories. We ran the benchmark with 20,000 files and 100,000 transactions, and with unbuffered file I/O. Default values were chosen for all remaining options. The total time reported is obtained by summing the times of the individual instances.

The UNIX `make` program is typically used to build an application program from source code. It sequentially executes multiple instances of compilers and other utility programs. `Make` provides an option for executing subtasks in parallel, but we chose to run multiple sequential `make`s, rather than a single parallel `make`, in order to maximize the concurrency in the requests presented to the operating system. For our third workload, we ran an instance of `make` simultaneously on each processor, each building the arbitrarily chosen GNU Flex program. We created one build directory for each processor and invoked a sequential `make` in each of these directories in parallel. All the builds shared a common source tree. A driver application synchronized the invocations to ensure simultaneous start times, tracked the runtime 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 for these experiments.

Figures 14, 15, and 16 show performance for each of the three benchmarks for both K42 and Linux. For SDET (Figure 14), we show two curves for K42, one using the original nondistributed (shared) implementations of all K42 memory-management objects and another using the distributed (locality-optimized) implementations. For the other two workloads (Figures 15 and 16), we show only distributed-implementation results. The SDET curves show raw

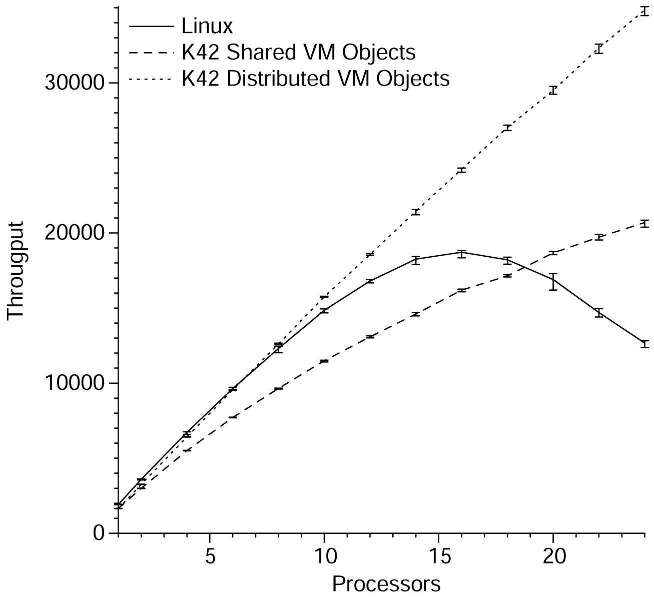

Fig. 14. Throughput of SDET-inspired benchmark.

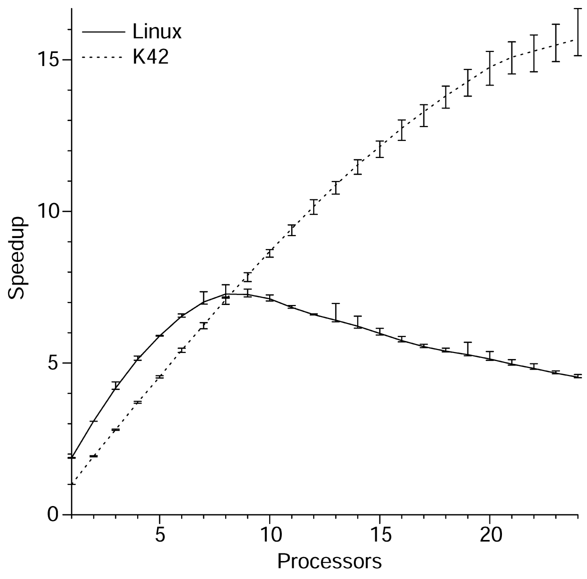

Fig. 15. Speedup of p independent instances of Postmark normalized to K42 uniprocessor result.

scripts-per-hour throughput, while the Postmark and `make` results are normalized to K42's execution times on one processor.

5.1.2 *Discussion.* The performance of Linux on these benchmarks (at the time we were making our measurements) illustrates the fact that a multiuser operating system can exhibit scalability problems even for workloads

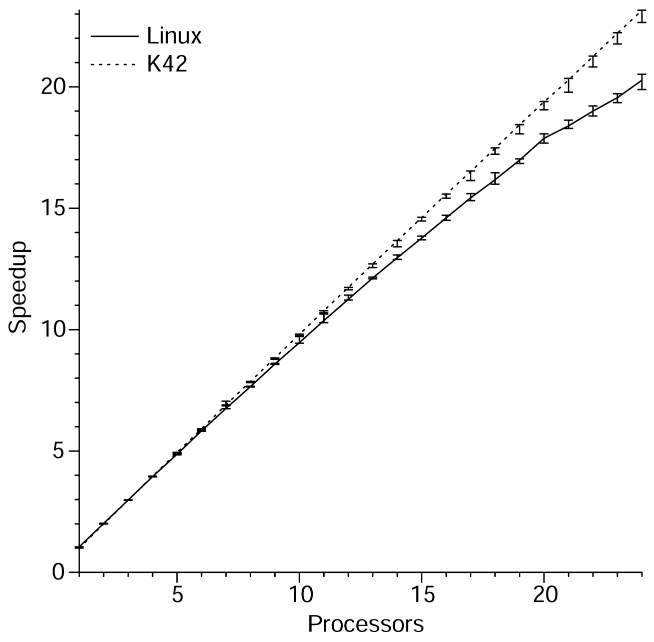

Fig. 16. Speedup of p independent instances of Make normalized to K42 uniprocessor result.

that are inherently scalable. Hence, developing models that help with improving scalability is relevant for even modest-sized systems running nonscientific workloads.

The figures show that K42 scales reasonably well for these workloads. Moreover, we see that K42's uniprocessor performance is comparable to that of Linux—an important result, because this kind of OO design and code reuse is thought to sacrifice base performance. Reasonable uniprocessor performance also lends legitimacy to the scalability results. (Scaling is easier if you start with bad uniprocessor numbers.)

For simultaneous independent sequential applications, K42's OO design all by itself provides a scalability advantage over more monolithic designs. However, the SDET results show that using an object-oriented approach alone is not sufficient for good scalability. Even though the request streams from the SDET multiuser workload are independent, there are still resources that are shared (common executable and data files, for example). Compare the K42 "Shared" and "Distributed" curves (the dashed and dotted lines) in Figure 14. Switching to distributed, locality-optimized implementations of the core objects of K42's virtual memory system leads to considerable improvement in scalability, getting us closer to the ideal in which many users can utilize the resources of the system independently.

It is important to note that these results are not intended as statements about either OS's absolute performance on these benchmarks. The results are a snapshot of the performance of the two systems at a particular point in time. Operating systems are not static, and their performance naturally improves as problems are identified and optimization efforts are made. An important aspect of the Clustered Object model is its ability to facilitate incremental optimization.

By the time this article was published, Linux had progressed to version 2.6. Changes to improve scalability were incorporated (although not much of the effort applies to the PowerPC virtual memory management code). Typical of the x86-centric efforts to improve scalability is the work described in Bryant et al. [2004], in which optimizations, similar to ours, are applied to the Linux VMM, but in an ad hoc manner. The effort expended on Linux to improve concurrency may ultimately make addressing locality more difficult. As systems grow larger, and as processor performance continues to improve relative to that of memory, we expect optimizing for locality to become more and more important. New structures and techniques, such as those used in K42, will be critical to achieving good locality, and therefore good scalability, in future systems.

5.2 Software Engineering Experience

Given our previous experience with SMMP operating systems, the simplicity of some of the object designs we have found sufficient was a surprise. While the simplicity results, in part, from a sophisticated infrastructure (e.g., memory allocator, IPC services, garbage collection strategy), we believe that it is mainly attributable to the underlying object-oriented design of the system. In previous systems, we had to develop a complicated implementation for each performance-critical service, because a single implementation had to perform well for a wide variety of workloads. In K42, individual object implementations are designed to handle specific workloads. This customizability led to a separation of concerns and allowed us to focus implementations on smaller sets of requirements.

When we first obtained our 24-processor system we found that our scalability was terrible. We were able to achieve reasonable scalability with about two weeks' work. We were able to progress rapidly because:

(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 could be encapsulated within objects. There were no cases where changes to a data structure resulted in changes propagating to other parts of the system.
(3) Deferred object deletion (i.e., the RCU protocol [McKenney et al. 2002]) eliminates the need for most existence locks and hence locking hierarchies. We found no cases where we had to modify complex locking protocols that permeated multiple objects.
(4) The OO design allowed us to develop special-purpose object implementations to deal with some of the hard cases without polluting our common-case implementations.
(5) Tracing infrastructure, which we had incorporated from the beginning, allowed us to categorize and correlate performance problems, such as lock contention, on an object-instance basis.

While the K42 OO design has been very useful for our research, we found that the approach splinters both the knowledge and implementation of system paths, making reasoning about and designing global policies more difficult. To date we have not found this concern to be prohibitive with respect to performance, but have found that the decomposition can be a barrier to system comprehension for new developers.

Specifically, there are a number of challenges presented by such a design. First, as more and more implementations are developed, interface changes and bug fixes might result in maintenance complexity. While we use inheritance aggressively in the system, in many cases (e.g., nondistributed versus distributed implementations) code cannot be shared easily. We are looking at various technical solutions to this problem.

Second, it can be difficult to ascertain a global state if all the data for achieving that understanding is scattered across many object instances. For example, without a central page cache, there is no natural way to implement a global clock algorithm for paging, although so far we have found that alternative algorithms (e.g., working set) are feasible and adequate for the workloads we have studied.

Third, while it is often possible to determine at instantiation time what object type is best suited for a particular workload, in many cases the knowledge is only available dynamically and may change over time. We have developed the hot-swapping services in K42 to allow objects to be changed on the fly, but it is not yet clear if this approach is sufficient.

Finally, we have noticed it is very difficult for new developers to understand the system, since so much of it is in infrastructure. While individual objects are simpler, the larger picture may be more obfuscated.

6. RELATED WORK

Much of the research into multiprocessor operating systems has been concerned with how to support new, different, or changing requirements in OS services, specifically focusing on user-level models of parallelism, resource management, and hardware configuration. We will generically refer to this as support for flexibility. In contrast, our research has pursued a performance-oriented approach. Figure 17 pictorially places our work on K42 and Clustered Objects (both described in Section 2) in context with prior work.

6.1 Early Multiprocessor OS Research Experience

Arguably the most complex computer systems are those with multiple processing units. The advent of multiprocessor computer systems present operating system designers with four intertwined issues:

(1) true parallelism (as opposed to just concurrency),
(2) new and more complex hardware features, such as multiple caches, multistaged interconnects, and complex memory and interrupt controllers,
(3) subtle and sensitive performance characteristics, and

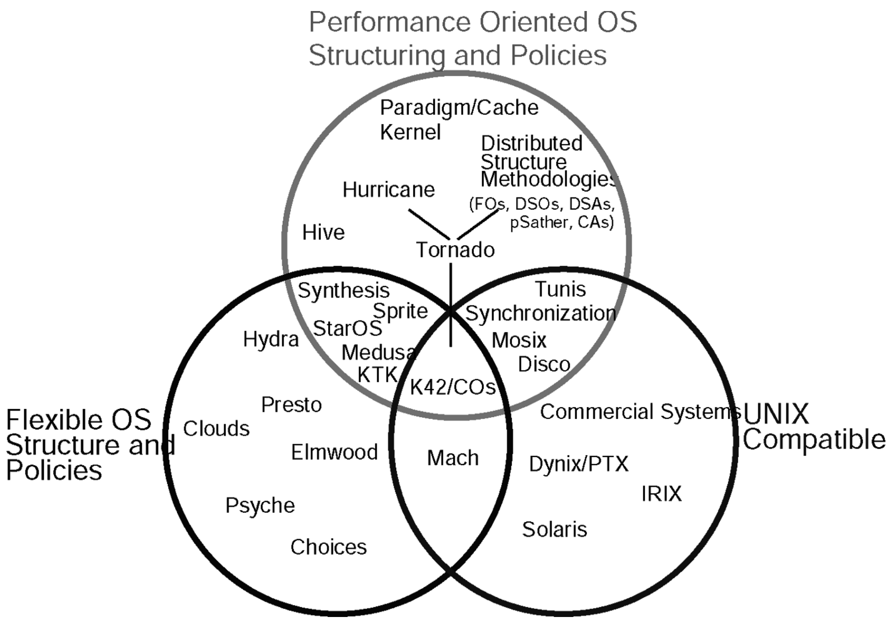

Fig. 17. Illustration of related work with respect to K42 and Clustered Objects

(4) the demand to facilitate user exploitation of the system's parallelism while providing standard environments and tools.

Industrial work on SMMP operating systems can be summarized as a study into how to evolve standard uniprocessor operating systems with the introduction of synchronization primitives [Denham et al. 1994; Kelley 1989; Kleiman et al. 1992; LoVerso et al. 1991; Lycklama 1985; McCrocklin 1995; Peacock et al. 1992; Russell and Waterman 1987]. This ensures correctness and permits higher degrees of concurrency in the basic OS primitives. The fundamental approach taken was to apply synchronization primitives to the uniprocessor code base in order to ensure correctness. Predominantly, the primitive adopted was a shared-memory lock, implemented on top of the atomic primitives offered by the hardware platform. The demand for higher performance led to successively finer-grain application of locks to the data structures of the operating systems. Doing so increased concurrency in the operating system at the expense of considerable complexity and loss of platform generality.

In contrast to the industrial research work, the majority of the early academic research work focused on flexibility and improved synchronization techniques. Systems that addressed flexibility include: Hydra [Levin et al. 1975; Wulf et al. 1975], StarOS [Cohen and Jefferson 1975], Medusa [Ousterhout et al. 1980], Choices [Campbell et al. 1993], Elmwood [Leblanc et al. 1989], Presto [Bershad et al. 1988], Psyche [Scott et al. 1988], Clouds [Dasgupta et al. 1991]. With the exception of Hydra, StarOS and Medusa, very few systems actually addressed unique multiprocessor issues or acknowledged specific multiprocessor implications on performance. In the remainder of this section we highlight work that either resulted in relevant performance observations or attempted to account for multiprocessor performance implications in operating system construction.

In 1985 the Tunis operating system was one of the first systems to focus on the importance of locality rather than flexibility [Ewens et al. 1985]. One of the aims of the project was to explore the potential for cheap multiprocessors systems, constructed from commodity single board microprocessors interconnected via a standard backplane bus. Although limited in nature, Tunis was one of the first operating systems to provide uniprocessor UNIX compatibility while employing a novel internal structure.

The early 1980s not only saw the emergence of tightly coupled, shared-memory multiprocessor systems such as the CM* [Ousterhout et al. 1980], but also loosely coupled distributed systems composed of commodity workstations interconnected via local area networking. Projects such as V [Cheriton and Zwaenepoel 1983] and Accent [Rashid 1986] attempted to provide a unified environment for constructing software and managing the resources of a loosely coupled distributed system. Unlike the operating systems for the emerging shared-memory multiprocessors, operating systems for distributed systems could not rely on hardware support for sharing. As such, they typically were constructed as a set of autonomous lightweight independent uniprocessor operating systems that cooperated via network messages to provide a loosely coupled unified environment. Although dealing with very different performance tradeoffs, the distributed systems work influenced and intertwined with SMMP operating systems research over the years. For example one of the key contributions of V was microkernel support of lightweight user-level threads that were first-class and kernel visible.

In the mid 1980s, the Mach operating system was developed at Carnegie Mellon University based on the distributed systems Rig and Accent [Rashid et al. 1988; Young et al. 1987]. One of the key factors in Mach's success was the early commitment to UNIX compatibility while supporting user-level parallelism. In spirit, the basic structure of Rig, Accent, and Mach is similar to Hydra and StarOS. All these systems are built around a fundamental IPC (Inter-Process Communication) model. Mach takes the traditional approach of fine-grain locking of centralized data structures to improve concurrency on multiprocessors.[16] The later Mach work provides a good discussion of the difficulties associated with fine-grain locking, covering issues of existence, mutual exclusion, lock hierarchies, and locking protocols [Black et al. 1991].

The MOSIX researchers have similarly observed the need to limit and bound communication when tackling the problems of constructing a scalable distributed system [Barak and La'adan 1998; Barak and Wheeler 1989]. MOSIX focuses on the issues associated with scaling a single UNIX system image to a large number of distributed nodes. The MOSIX work strives to ensure that the design of the internal management and control algorithms imposes a fixed amount of overhead on each processor, regardless of the number of nodes in the system. Probabilistic algorithms are employed to ensure that all kernel

[16]Industrial systems such as Sequent's Dynix [Beck and Kasten 1985; Garg 1990; Inman 1985], one of Mach's contemporaries, employed fine-grain locking.

interactions involve a limited number of processors and that the network activity is bounded at each node.

Cheriton et al. [1991] proposed an aggressive, distributed shared-memory parallel hardware architecture called Paradigm and also described OS support for it based on multiple cooperating instances of the V microkernel, a simple hand-tuned kernel designed for distributed systems construction, with the majority of OS function implemented as user-level system servers. The primary approach to supporting sharing and application coordination on top of the multiple microkernel instances was through the use of a distributed file system. The authors state abstractly that kernel data structures, such as dispatch queues, are to be partitioned across the system to minimize interprocessor interference and to exploit efficient sharing, using the system's caches. Furthermore, they state that cache behavior is to be taken into account by using cache-friendly locking and data structures designed to minimize misses, with cache alignment taken into consideration. Finally they also assert that the system will provide UNIX emulation with little performance overhead. It is unclear to what extent the system was completed.

In 1994, as part of the Paradigm project, an alternative OS structure dubbed the Cache Kernel was explored by Cheriton et al. [Cheriton and Duda 1994]. At the heart of the Cache Kernel model was the desire to provide a finer-grain layering of the system, where user-level application kernels are built on top of a thin cache kernel that only supports basic memory mapping and trap reflection facilities via an object model. From a multiprocessor point of view, however, its architecture remained the same as the previous work, where a cluster of processors of the Paradigm system ran a separate instance of a Cache Kernel.

The RP3 project attempted to use the Mach microkernel to enable multiprocessor research [Bryant et al. 1991]. The RP3 authors found bottlenecks in both the UNIX and Mach, code with congestion in memory modules being the major source of slowdown. To reduce contention, the authors used hardware-specific memory interleaving, low-contention locks, and localized free lists. We contend that the same benefits could have been achieved if locality had been explicitly exploited in the basic design. UNIX compatibility and performance were critical to the RP3 team. In the end, the flexibility provided by Mach did not seem to be salient to the RP3 researchers. Mach's internal traditional shared structure limited performance and its flexibility did not help to address these problems.

6.2 Distributed Data Structures

In this subsection we focus on the research related to the use of distributed data structures.

6.2.1 *Distributed Systems: FOs and DSOs.* Fragmented Objects(FOs) [Brun-Cottan and Makpangou 1995; Makpangou et al. 1994; Shapiro et al. 1989] and Distributed Shared Objects (DSOs) [Bal et al. 1989; Homburg et al. 1995; van Steen et al. 1997] both explore the use of a partitioned object model as a programming abstraction for coping with the latencies in a distributed network environment. Fragmented Objects represent an object as a set of fragments that exist in address spaces distributed across the machines of a local area network, while the object appears to the client as a single entity. When a client invokes a method of an object, it does so by invoking a method of a fragment local to its address space. The fragments, transparently to the client, communicate amongst themselves to ensure a consistent global view of the object. The Fragmented Objects work focuses on how to codify flexible consistency protocols within a general framework.

In the case of Distributed Shared Objects, distributed processes communicate by accessing a distributed shared object instance. Each instance has a unique id and one or more interfaces. In order to improve performance, an instance can be physically distributed with its state partitioned or replicated across multiple machines at the same time. All protocols for communication, replication, distribution, and migration are internal to the object and hidden from clients. A coarse-grain model of communication is assumed, focusing on the nature of wide area network applications and protocols such as that of the World Wide Web. For example, global uniform naming and binding services have been addressed.

Clustered Objects are similar to FOs and DSOs in that they distribute/replicate state but hide this from clients by presenting the view of a single object. Clustered Object representatives correspond to FOs fragments.

6.2.2 *Language Support: CAs and pSather.* Chien et al. introduced Concurrent Aggregates (CAs) as a language abstraction for expressing parallel data structures in a modular fashion [Chien and Dally 1990]. This work is concerned with the language issues of supporting a distributed parallel object model for efficient construction of parallel applications in a message-passing environment. Similar to several concurrent object-oriented programming systems, an Actor model [Agha 1990] is adopted. An instance of an Aggregate has a single external name and interface, however, each invocation is translated by a runtime environment to an invocation on an arbitrary representative of the Aggregate. The number of representatives for an Aggregate is declared by the programmer as a constant. Each representative contains local instances of the Aggregate fields. The language supports the ability for one representative of an Aggregate to name/locate and invoke methods of the other representatives in order to permit scatter and gather operations via function shipping and more complex cooperation.

pSather explores language and associated runtime extensions to support data distribution on NUMA multiprocessors for Sather, an Eiffel-like research language [Lim 93]. Specifically, pSather adds threads, synchronization, and data distribution to Sather. Unlike the previous work discussed, pSather advocates orthogonality between object-orientation and parallelism; it introduces new language constructs independent of the object model for data distribution. Unlike Chien's Concurrent Aggregates, it does not impose a specific processing/synchronization model, nor does it assume the use of system-provided consistency models/protocols such as Distributed Shared Objects or Fragmented Objects. In his thesis, Lim proposes two primitives for replicating reference variables and data structures such that a replica is located in each cluster of a NUMA multiprocessor. Since the data distribution primitives are not integrated into the object model, there is no native support for hiding the distribution behind an interface. In contrast to our work, there is no support for dynamic instantiation or initialization of replicas, nor facilities for distributed reclamation.

Although the primary focus of the pSather work is on evaluating the overall programmer experience, using the parallel primitives and the library of data structures developed, the performance of the applications is also evaluated with respect to scale. The author highlights some of the trade-offs in performance with respect to remote accesses given variations in the implementation of the data structures and algorithms of the applications. He points out that minimizing remote accesses, thus enhancing locality, is key to good performance for the applications studied.

6.2.3 *Topologies and DSAs.* In the early 1990s there was considerable interest in message-passing architectures, which typically leveraged a point-to-point interconnection network and the promise of unlimited scalability. In an attempt to ease the burden and generalize the use of such machines, Schwan and Bo [1990] proposed OS support for a distributed primitive called a Topology. A Topology attempts to isolate and encapsulate the communication protocol and structure among a number of identified communicating processes via a shared object-oriented abstraction. A Topology's structure is described as a set of vertices and edges where the vertices are mapped to physical nodes of the hardware and edges capture the communication structure between pairs of nodes corresponding to vertices. The Topology is implemented to minimize the number of nonlocal communication for the given architecture being used. They are presented as heavyweight OS abstractions requiring considerable OS support for their management and scheduling.

In subsequent work, motivated by significant performance improvements obtained with distributed data structures and algorithms on a NUMA multiprocessor for Traveling Sales Person (TSP) programs [Mukherjee and Schwan 1993], Clémençon et al. [1993, 1996] proposed a distributed object model called Distributed Shared Abstractions (DSAs). This work is targeted at increasing the scalability and portability of parallel programs via a reusable user-level library that supports the construction of objects that encapsulate a DSA. Each object is composed of a set of distributed fragments similar to the Fragmented Objects and Distributed Shared Objects discussed earlier. The runtime implementation and model are, however, built on the previous work on Topologies.

Clémençon et al. observed two factors that affect performance of shared data structures on a NUMA multiprocessor: (1) contention due to concurrent access (synchronization overhead), and (2) remote memory access costs (communication overhead).

They observed that distribution of state is key to reducing contention and improving locality. When comparing multiple parallel versions of the TSP program, they found that using a centralized work queue protected by a single spin lock limited speedup to a factor of 4, whereas a factor of 10 speedup was possible with a distributed work queue on a system with 25 processors. Further, they found that, by leveraging application-specific knowledge, they were able to specialize the distributed data structure implementation to further improve performance. They demonstrated that despite additional complexities, distributed implementations with customized semantics can significantly improve application performance. Note that in the results presented Clémençon et al. do not isolate the influence of synchronization overhead versus remote access.[17]

6.2.4 *Hash Tables.* There has been a body of work that has looked at the use of distributed hash tables in the context of specific distributed applications, including distributed databases [Ellis 1983], cluster-based Internet services [Gribble et al. 2000], peer-to-peer systems [Dabek et al. 2001], and general distributed data storage and lookup services [Cates 2003]. In general, the work done on distributed data structures for distributed systems is primarily concerned with the exploration of the distributed data structure as a convenient abstraction for constructing network-based applications, increasing robustness via replication. Like the Fragmented Object [Brun-Cottan and Makpangou 1995; Makpangou et al. 1994; Shapiro et al. 1989] and Distributed Shared Object [Bal et al. 1989; Homburg et al. 1995; van Steen et al. 1997] work, the use of distributed hash tables seeks a systematic way of reducing and hiding network latencies. The coarse-grain nature and network focus of this type of work results in few insights for the construction of performance-critical and latency-sensitive shared-memory multiprocessor systems software.

With respect to motivation and purpose, our hash table work is similar to the software set-associative cache architecture described in Peacock et al. [1992], which was developed in the context of multithreading a traditional UNIX system kernel. To avoid overheads associated with fine-grain locking, we developed an alternative hash structure to serve the specific requirements of operating system caches, trying to avoid synchronization. Our work not only differs significantly with respect to implementation but also in our focus on locality. We attempt to eliminate the need for synchronization on hot paths through the use of distribution.

6.3 Modern Multiprocessor Operating Systems Research

A number of papers have been published on performance issues in shared-memory multiprocessor operating systems, but mostly in the context of resolving specific problems in a specific system [Campbell et al. 1991; Chapin et al. 1995; Cheriton and Duda 1994; McCrocklin 1995; Presotto 1990; Talbot 1995]. These operating systems were mostly for small-scale multiprocessor systems, trying to scale up to larger systems. Other work on locality issues in operating system structure was mostly either done in the context of earlier non–cache-coherent NUMA systems [Chaves, Jr. et al. 1993], or, as in the case of Plan 9, was not published [Pike 1998]. Two projects that were aimed explicitly at large-scale multiprocessors were Hive [Chapin et al. 1995] and Hurricane [Unrau et al. 1995]. Both independently chose a clustered approach by connecting multiple small-scale systems to form either, in the case of Hive, a more fault-tolerant

[17]Concurrent centralized implementations that employ fine-grain locking or lock-free techniques were not considered.

system, or, in the case of Hurricane, a more scalable system. However, both groups ran into complexity problems with this approach and both have moved on to other approaches; namely Disco [Bugnion et al. 1997] and Tornado [Gamsa 1999], respectively.

Other more coarse-grain approaches for improving locality in general SMMP software include automated support for memory page placement, replication, and migration [LaRowe, Jr. and Ellis 1991; Marchetti et al. 1995; Verghese et al. 1996] and cache-affinity-aware process scheduling [Devarakonda and Mukherjee 1991; Gupta et al. 1991; Markatos and LeBlanc 1994; Squillante and Lazowska 1993; Vaswani and Zahorjan 1991].

6.3.1 *Operating Systems Performance.* Poor performance of the operating system can have considerable impact on application performance. For example, for parallel workloads studied by Torrellas et al., the operating system accounted for as much as 32% to 47% of the nonidle execution time [Torrellas et al. 1992]. Similarly Xia and Torrellas [1996] showed that for a different set of workloads, 42% to 54% of the time was spent in the operating system, while Chapin et al. [1995] found that 24% of total execution time was spent in the operating system for their workload.

The traditional approach to developing SMMP operating systems has been to start with a uniprocessor operating system and then successively tune it for concurrency. This is achieved by adding locks to protect critical resources. Performance measurements are then used to identify points of contention. As bottlenecks are identified, locks are split into multiple locks to increase concurrency, leading to finer-grain locking. Several commercial SMMP operating systems have been developed as successive refinements of a uniprocessor code base. Denham et al. [1994] provides an excellent account of one such development effort. This approach is ad hoc in nature and leads to complex systems, while providing little flexibility. Adding more processors to the system, or changing access patterns, may require significant retuning.

The continual splitting of locks can also lead to excessive locking overheads. In such cases, it is often necessary to design new algorithms and data structures that do not depend so heavily on synchronization. Examples include: software setassociative cache architecture [Peacock et al. 1992]; kernel memory allocation facilities [McKenney and Slingwine 1993]; fair, fast, scalable reader-writer locks [Krieger et al. 1993]; performance-measurement kernel device driver [Anderson et al. 1997]; and intranode data structures [Stets et al. 1997].

The traditional approach of splitting locks and selectively redesigning also does not explicitly lead to increased locality. Chapin et al. [1995] studied the memory system performance of a commercial UNIX system, parallelized to run efficiently on the 64-processor Stanford DASH multiprocessor. They found that the time spent servicing operating system data misses was three times the time spent executing operating system code. Of the time spent servicing operating system data misses, 92% was due to remote misses. Kaeli et al. [1997] showed that careful tuning of their operating system to improve locality allowed them to obtain linear speedups on their prototype CC-NUMA system, running OLTP benchmarks.

In the early to mid 1990s, researchers identified memory performance as critical to system performance [Chapin et al. 1995; Chen and Bershad 1993; Maynard et al. 1994; Rosenblum et al. 1995; Torrellas et al. 1992]. They noted that cache performance and coherence are critical aspects of SMMP hardware that must be taken into account by software, and that focusing on concurrency and synchronization is not enough.

Rosenblum et al. [1995] explicitly advocated that operating systems be optimized to meet the demands of users for high performance. However, they point out that operating systems are large and complex and the optimization task is difficult and, without care, tuning can result in increased complexity with little impact on the end-user performance. The key is to focus on optimization by identifying performance problems. They studied three important workloads: (1) program development workload, (2) database workload, and (3) large simulations that stress the memory subsystem.

They predicted that, even for small scale SMMPs, coherence overheads induced by communication and synchronization overheads would result in MP OS services consuming 30% to 70% more resources than uniprocessor counterparts. They also observed that larger caches do not help alleviate coherence overhead, so the performance gap between MP OSes and UP OSes will grow unless there is focus on kernel restructuring to reduce unnecessary communication. They pointed out that, as the relative cost of coherence misses goes up, programmers must focus on data layout to avoid false sharing, and that preserving locality in scheduling is critical to ensuring effectiveness of caches. Rescheduling processes on different processors can result in coherence traffic on kernel data structures.

Unlike many of the previously discussed MP OS research efforts, the work from University of Toronto chose to first focus on multiprocessor performance, thereby motivating, justifying, and evaluating the operating system design and implementation based on the structure and properties of scalable multiprocessor hardware. Motivated by the Hector multiprocessor [Vranesic et al. 1991], representative of the architectures for large-scale multiprocessors of the time [BBN Advanced Computers, Inc. 1988; Frank et al. 1993; Lenoski et al. 1992; Pfister et al. 1985], the group chose a simple structuring for the operating system that directly mirrored the architecture of the hardware, hoping to leverage the strengths of the hardware structure while minimizing its weaknesses.

By focusing on performance rather than flexibility, the Hurricane group was motivated to acknowledge, analyze, and identify the unique operating system requirements with respect to scalable performance. Particularly, based on previous literature and queuing theory analysis, the following guidelines were identified [Unrau et al. 1995]:

Preserving parallelism: The operating system must preserve the parallelism afforded by the applications. If several threads of an executing application (or of independent applications running at the same time) request independent operating system services in parallel, they must be serviced in parallel; otherwise the operating system becomes a bottleneck, limiting scalability and application speedup.

Bounding overhead: The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors. If the overhead of each service call increases with the number of processors, the system will ultimately saturate, so the demand on any single resource cannot increase with the number of processors. For this reason, system-wide ordered queues cannot be used, and objects cannot be located by linear searches if the queue lengths or search lengths increase with the size of the system. Broadcasts cannot be used for the same reason.

Preserving locality: The operating system must preserve the locality of the applications. Specifically it was noted that locality can be increased (*i*) by properly choosing and placing data structures within the operating system, (*ii*) by directing requests from the application to nearby service points, and (*iii*) by enacting policies that increase locality in the applications' memory accesses.

Although some of these guidelines have been identified by other researchers [Barak and Kornatzky 1987; Smith 1994], we are not aware of other general-purpose shared-memory multiprocessor operating systems that pervasively utilize them in their design. Over the years, these guidelines have been refined but have remained a central focus of our body of research.

Hurricane, in particular, employed a coarse-grain approach to scalability in which a single large-scale SMMP was partitioned into clusters of a fixed number of processors. Each cluster ran a separate instance of a small-scale SMMP operating system, cooperatively providing a single system image. Hurricane attempted to directly reflect the hardware structure, utilizing a collection of separate instances of a small-scale SMP operating system. Despite many of the positive benefits of clustering, it was found that: (*i*) the traditional intracluster structures exhibit poor locality, which severely impacts performance on modern multiprocessors; (*ii*) the rigid clustering results in increased complexity as well as high overhead or poor scalability for some applications; (*iii*) the traditional structures as well as the clustering strategy make it difficult to support the specialized policy requirements of parallel applications [Gamsa 1999].

The work at Stanford on the Hive operating system [Chapin et al. 1995] also focused on clustering, first as a means of providing fault containment and second as a means for improving scalability. Having experienced similar complexity and performance problems with the use of fixed clustering, the Stanford research group began a new project in the late 1990s called Disco [Bugnion et al. 1997; Govil et al. 1999]. The Disco project pursued strict partitioning as a means for leveraging the resources of a multiprocessor. Rather than trying to construct a kernel that can efficiently support a single system image, they pursue the construction of a kernel that can support the execution of multiple virtual machines (VMs). By doing so, the software within the virtual machines is responsible for extracting the degree of parallelism it requires from the resources allocated to the VM on which it is executing. Rather than wasting the resources of a large-scale machine on a single OS instance that is incapable of efficiently utilizing all the resources, the resources are partitioned across multiple OS instances. There are three key advantages to this approach:

(1) The underlying systems software that enables the partitioning does not itself require high concurrency.
(2) Standard workloads can be run by leveraging the Virtual Machine approach to run standard OS instances.
(3) Resources of a large-scale machine can be efficiently utilized with standard software, albeit without native support for large-scale applications and with limited sharing between partitions.

To some extent this approach can be viewed as a trade-off that permits large-scale machines to be leveraged using standard systems software.

Like the Stanford group, the Toronto researchers also pursued a new project based on their experience with fixed clustering. In contrast, however, the Toronto group chose to pursue an operating systems structure that relaxed the boundaries imposed by clustering when constructing its new operating system, called Tornado. The fundamental approach was to explore the structuring of an operating system kernel so that a single system image could be efficiently scaled without having to appeal to the fixed boundaries of clustering.

Like other systems, Tornado had an object-oriented design, but not primarily for the software engineering benefits or for flexibility, but rather for multiprocessor performance benefits. Details of the Tornado operating system are given elsewhere [Gamsa et al. 1999; Gamsa 1999]. The contributions of the Tornado work include:

(1) an appropriate object decomposition for a multiprocessor operating system,
(2) scalable and efficient support in the object runtime that would enable de-clustered (distributed) implementations of any object (Objects in Tornado were thus dubbed Clustered Objects),
(3) a semi-automatic garbage collection scheme incorporated in the object runtime system that facilitates localizing lock accesses and greatly simplifies locking protocols,[18] and
(4) a core set of low-level operating system facilities that are tuned for multiprocessor performance and show high degrees of concurrency and locality.

The key low-level OS facilities on which Tornado focused were:

—Scalable, efficient multiprocessor memory allocation.
—Light-weight protection-domain crossing that is focused on preserving locality, utilizing only local processor resources in the common case.

In contrast with the work presented in this paper, the majority of the system objects in Tornado did not utilize distribution. Gamsa's work on Clustered Objects in Tornado focused on developing the underlying infrastructure and basic

[18]With the garbage collection scheme, no additional (existence) locks are needed to protect the locks internal to the objects. As a result, Tornado's *locking strategy* results in much lower locking overhead, simpler locking protocols, and can often eliminate the need to worry about lock hierarchies. As part of Tornado's garbage collection scheme, the Toronto research group independently developed a lock-free discipline similar to that of Read-Copy-Update [McKenney and Slingwine 1998].

mechanisms [Gamsa 1999]. The work presented in this paper, developing the Clustered Object model and studying the development and use of distributed object implementations, began in Tornado [Appavoo 1998] utilizing the supporting mechanisms developed by Gamsa.

The work described in Bryant et al. [2004] in improving Linux scalability explores many locality optimizations in an ad hoc manner. Many fixes discussed in the paper localize memory accesses on hot paths and use scatter/gather operations on the less performance-sensitive paths. The authors point out that relocating data into perprocessor storage is not a "panacea" but must be balanced with appropriate use of global shared-memory accesses. Our work provides a framework for developing software in this fashion.

7. CONCLUDING REMARKS

There is a pervasive belief that operating systems are fundamentally unable to scale to large SMMPs except for specialized scientific applications with limited requirements for OS functionality. There is also a widespread belief that any operating system that has reasonable scalability will exhibit poor base performance and extreme complexity, making it inappropriate for general systems and workloads. These beliefs, based on current OS structures and performance, have had a large impact on how existing commercial hardware systems developed by IBM and others are constructed. In this work, we demonstrated that a fully partitioned and locality-optimized implementation of a core OS service, virtual memory management, is possible. The existence proof for such a distributed, high-performance implementation is, we believe, a major research result.

We described how we distributed a number of key objects in K42 and showed the effect of this distribution on SMMP performance, using both microbenchmarks and application benchmarks. The purpose of these distributions is to optimize access to shared data on critical system paths. The techniques we described are indicative of the approach we have taken across the whole system as we identified performance problems. We believe that the object-oriented design, and the infrastructure we developed to support it, greatly reduce the complexity required to achieve good scalability. It allows the developer to focus on specific objects, without having to worry about larger system-wide protocols, enabling an incremental optimization strategy. Objects are specialized to handle specific demands, leading to a separation of concerns and simplifying the development of distributed implementations.

The sophisticated infrastructure we developed in K42, and the global use of OO design, will likely not be adopted widely. However, while we believe that such a design is critical for scaling to hundreds of processors, ideas from K42 can be adopted incrementally for systems targeting smaller machines. For example, K42 has a very aggressive implementation of read-copy-update [McKenney and Slingwine 1998], and exploits state available from our object structure to make it scalable. The same technology has been transferred to Linux and resulted in performance gains in 4- and 8-way systems. Similarly, Linux adopted from

K42 an object-oriented approach to maintaining reverse mappings for memory management. We believe that more OO structure can be incorporated into Linux and other commercial systems, and that the use of OO design will enable the kind of distributed implementations we have found useful in K42 to be used in Linux as well.

K42 is available under the LGPL license, and is jointly being developed by IBM and a number of universities.

REFERENCES

AGHA, G. 1990. Concurrent object-oriented programming. *Comm. ACM 33*, 9, 125–141.

ANDERSON, J. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T. A., SITES, R. L., VANDERVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. 1997. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-16)*. ACM Press, 1–14.

APPAVOO, J. 1998. Clustered Objects: Initial design, implementation and evaluation. M.S. thesis, Department of Computing Science, University of Toronto, Toronto, Canada.

APPAVOO, J. 2005. Clustered objects. Ph.D. thesis, University of Toronto, Toronto, Canada.

APPAVOO, J., AUSLANDER, M., EDELSOHN, D., SILVA, D. D., KRIEGER, O., OSTROWSKI, M., ROSENBURG, B., WISNIEWSKI, R. W., AND XENIDIS, J. 2003. Providing a linux API on the scalable K42 kernel. In *Freenix*. San Antonio, TX.

APPAVOO, J., HUI, K., STUMM, M., WISNIEWSKI, R. W., DA SILVA, D., KRIEGER, O., AND SOULES, C. A. N. 2002. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*. ACM Press, 3–8.

BAL, H. E., KAASHOEK, M. F., AND TANENBAUM, A. S. 1989. A distributed implementation of the shared data-object model. In *Proceedings of the 1st USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*. E. Spafford, Ed. Ft. Lauderdale FL, 1–19.

BARAK, A. AND KORNATZKY, Y. 1987. Design principles of operating systems for large scale multicomputers. Tech. rep., IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY.

BARAK, A. AND LA'ADAN, O. 1998. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generat. Comput. Syst. 13*, 4–5, 361–372.

BARAK, A. AND WHEELER, R. 1989. MOSIX: An integrated multiprocessor UNIX. In *Proceedings of the Winter 1989 USENIX Conference:* San Diego, CA. USENIX, Berkeley, CA. 101–112.

BAUMANN, A., APPAVOO, J., SILVA, D. D., KERR, J., KRIEGER, O., AND WISNIEWSKI, R. W. 2005. Providing dynamic update in an operating system. In *USENIX Technical Conference*. Anaheim, CA, 279–291.

BBN Advanced Computers, Inc. 1988. *Overview of the Butterfly GP1000*. BBN Advanced Computers, Inc.

BECK, B. AND KASTEN, B. 1985. VLSI assist in building a multiprocessor UNIX system. In *Proceedings of the USENIX Summer Conference.* Portland, OR. USENIX, 255–275.

BERSHAD, B. N., LAZOWSKA, E. D., LEVY, H. M., AND WAGNER, D. B. 1988. An open environment for building parallel programming systems. In *Proceedings of the ACM/SIGPLAN Conference on Parallel Programming: Experience with Applications, Languages and Systems*. ACM Press, 1–9.

BLACK, D. L., TEVANIAN, JR., A., GOLUB, D. B., AND YOUNG, M. W. 1991. Locking and reference counting in the Mach kernel. In *Proceedings of the International Conference on Parallel Processing*. Vol. II, Software. CRC Press, Boca Raton, FL, II–167–II–173.

BRUN-COTTAN, G. AND MAKPANGOU, M. 1995. Adaptable replicated objects in distributed environments. Tech. rep. BROADCAST TR No.100, ESPRIT Basic Research Project BROADCAST.

BRYANT, R., CHANG, H.-Y., AND ROSENBURG, B. 1991. Operating system support for parallel programming on RP3. *IBM J. Res. Devel. 35*, 5/6 (Sept.), 617–634.

BRYANT, R., HAWKES, J., AND STEINER, J. 2004. Scaling linux to the extreme: from 64 to 512 processors. *Ottawa Linux Symposium*. Linux Symposium.

BUGNION, E., DEVINE, S., AND ROSENBLUM, M. 1997. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-16)*. ACM Press, 143–156.

CAMPBELL, M., BARTON, R., BROWNING, J., CERVENKA, D., CURRY, B., DAVIS, T., EDMONDS, T., HOLT, R., SLICE, J., SMITH, T., AND WESCOTT, R. 1991. The parallelization of UNIX system V release 4.0. In *USENIX Conference Proceedings*. USENIX, Dallas, TX, 307–324.

CAMPBELL, R. H., ISLAM, N., RAILA, D., AND MADANY, P. 1993. Designing and implementing Choices: An object-oriented system in C++. *Comm. ACM 36*, 9 (Sept.), 117–126.

CATES, J. 2003. Robust and efficient data management for a distributed hash table. M.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

CHAPIN, J., HERROD, S. A., ROSENBLUM, M., AND GUPTA, A. 1995. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'95/PERFORMANCE'95)*. 1–13.

CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. 1995. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*. ACM Press, 12–25.

CHAVES, JR., E. M., DAS, P. C., LEBLANC, T. J., MARSH, B. D., AND SCOTT, M. L. 1993. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Pract. Exper. 5*, 3 (May), 171–191.

CHERITON, D. R. , GOOSEN, H. A., AND BOYLE, P. D. 1991. Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Comput.* 24, 2 (Feb.), 33–46.

CHEN, J. B. AND BERSHAD, B. N. 1993. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*. ACM Press, 120–133.

CHERITON, D. R. AND DUDA, K. J. 1994. A Caching model of operating system kernel functionality. In *Operating Systems Design and Implementation*. 179–193.

CHERITON, D. R. AND ZWAENEPOEL, W. 1983. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP-9)*. ACM Press, 129–140.

CHIEN, A. A. AND DALLY, W. J. 1990. Concurrent Aggregates (CA). In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'90),* ACM SIGPLAN Notices. 187–196.

CLÈMENÇON, C., MUKHERJEE, B., AND SCHWAN, K. 1996. Distributed shared abstractions (DSA) on multiprocessors. *IEEE Trans. Softw. Engin.* 22, 2 (Feb.) 132–152.

CLÈMENÇON, C., MUKHERJEE, B., AND SCHWAN, K. 1993. Distributed shared Abstractions (DSA) on Large-scale multiprocessors. In *Proceedings of the Symposium on Experience with Distributed and Multiprocessor Systems.* USENIX, San Diego, CA, 227–246.

COHEN, E. AND JEFFERSON, D. 1975. Protection in the Hydra operating system. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP-5)*. ACM Press, 141–160.

DABEK, F., BRUNSKILL, E., KAASHOEK, M. F., KARGER, D., MORRIS, R., STOICA, I., AND BALAKRISHNAN, H. 2001. Building peer-to-peer systems with Chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. IEEE Computer Society.

DASGUPTA, P., LEBLANC, JR., R. J., AHAMAD, M., AND RAMACHANDRAN, U. 1991. The clouds distributed operating system. *IEEE Comput. 24*, 11 (Nov.), 34–44.

DENHAM, J. M., LONG, P., AND WOODWARD, J. A. 1994. DEC OSF/1 version 3.0 symmetric multiprocessing implementation. *Digital Tech. J. Digital Equip. Corpo. 6*, 3 (Summer), 29–43.

DEVARAKONDA, M. AND MUKHERJEE, A. 1991. Issues in implementation of cache-affinity scheduling. In *Proceedings of the Usenix Winter Technical Conference*. USENIX, Berkeley, CA, 345–358.

ELLIS, C. S. 1983. Extensible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM Press, 106–116.

EWENS, P., BLYTHE, D. R., FUNKENHAUSER, M., AND HOLT, R. C. 1985. Tunis: A distributed multiprocessor operating system. In *Summer conference proceedings*. Portland, OR. USENIX, Berkeley, CA, 247–254.

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

Gamma, E., Helm, R., and Johnson, R. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

Gamsa, B. 1999. Tornado: Maximizing locality and concurrency in a shared-memory multiprocessor operating system. Ph.D. thesis, University of Toronto, Toronto, Canada.

Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. 1999. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*. 87–100.

Gao, H., Groote, J. F., and Hesselink, W. H. 2005. Lock-free dynamic hash tables with open addressing. *Distrib. Comput. 18*, 1 (July), 27–42.

Garg, A. 1990. Parallel STREAMS: A multiprocessor implementation. In *Proceedings of the Winter USENIX Conference, Washington, DC.,* USENIX, Berkeley, CA, 163–176.

Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. 1999. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP-17)*. ACM Press, 154–169.

Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D. 2000. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*. USENIX, Berkeley, CA, 319–332.

Gupta, A., Tucker, A., and Urushibara, S. 1991. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Stanford Univ., San Diego, CA, 120.

Homburg, P., van Doorn, L., van Steen, M., Tanenbaum, A. S., and de Jonge, W. 1995. An object model for flexible distributed systems. In *1st Annual ASCI Conference*. Heijen, Netherlands, 69–78. http://www.cs.vu.nl/˜steen/globe/publications.html.

Hui, K., Appavoo, J., Wisniewski, R. W., Auslander, M., Edelsohn, D., Gamsa, B., Krieger, O., Rosenburg, B., and Stumm, M. 2001. Position summary: Supporting hot-swappable components for system software. In *HotOS*. IEEE Computer Society.

Inman, J. 1985. Implementing loosely coupled functions on tightly coupled engines. In *Proceedings of the Summer Conference,* Portland. OR USENIX, Berkeley, CA, 277–298.

Kaeli, D. R., Fong, L. L., Booth, R. C., Imming, K. C., and Weigel, J. P. 1997. Performance Analysis on a CC-NUMA prototype. *IBM J. Res. Devl.* 41, 3, 205.

Katcher, J. 1997. Postmark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance.

Kelley, M. H. 1989. Multiprocessor aspects of the DG/UX kernel. In *Proceedings of the Winter USENIX Conference:* San Diego, CA. USENIX, Berkeley, CA, 85–99.

Kleiman, S., Voll, J., Eykholt, J., Shivalingah, A., Williams, D., Smith, M., Barton, S., and Skinner, G. 1992. Symmetric multprocessing in Solaris. COMPCON, San Francisco, CA.

Krieger, O., Stumm, M., Unrau, R. C., and Hanna, J. 1993. A fair fast scalable reader-writer lock. In *Proceedings of the International Conference on Parallel Processing*. Vol. II—Software. CRC Press, Boca Raton, FL, II–201–II–204.

LaRowe, Jr., R. P. and Ellis, C. S. 1991. Page placement policies for NUMA multiprocessors. *J. Parall. Distrib. Comput. 11*, 2 (Feb.) 112–129.

Leblanc, T. J., Mellor-Crummey, J. M., Gafter, N. M., Crowl, L. A., and Dibble, P. C. 1989. The Elmwood multiprocessor operating system. *Softw. Prac. Exper. 19*, 11 (11), 1029–1056.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S. 1992. The Stanford Dash multiprocessor. *IEEE Comput. 25*, 3 (March), 63–80.

Levin, R., Cohen, E., Corwin, W., Pollack, F., and Wulf, W. 1975. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP-5)*. ACM Press, 132–140.

Liedtke, J., Elphinstone, K., Schoenberg, S., and Haertig, H. 1997. Achieved IPC performance. *The 6th Workshop on Hot Topics in Operating Systems: Cape Cod, MA.*, IEEE Computer Society Press, 28–31.

Lim, C.-C. 93. A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions. Tech. rep. TR-93-063, Oct. Berkeley University of California, CA.

LoVerso, S., Paciorek, N., Langerman, A., and Feinberg, G. 1991. The OSF/1 UNIX filesystem (UFS). In *USENIX Conference Proceedings*. USENIX, 207–218.

Lycklama, H. 1985. UNIX on a microprocessor—10 years later. In *Proceedings of the Summer Conference,* Portland, OR. USENIX, Berkeley, CA, 5–16.

Makpangou, M., Gourhant, Y., Narzul, J.-P. L., and Shapiro, M. 1994. Fragmented objects for distributed abstractions. In *Readings in Distributed Computing Systems*, T. L. Casavant and M. Singhal, Eds. IEEE Computer Society Press, Los Alamitos, CA, 170–186.

Marchetti, M., Kontothanassis, L., Bianchini, R., and Scott, M. 1995. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95)*. IEEE Computer Society Press, Los Alamitos, CA, 480–485.

Markatos, E. P. and LeBlanc, T. J. 1994. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. Parall. Distrib. Syst. 5*, 4 (Apr.), 379–400.

Maynard, A. M. G., Donnelly, C. M., and Olszewski, B. R. 1994. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *ACM SIGPLAN Notices 29*, 11 (Nov.), 145–156.

McCrocklin, D. 1995. Scaling Solaris for enterprise computing. In *CUG 1995 Spring Proceedings*. Cray User Group, Inc., Denver, CO, 172–181.

McKenney, P. E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., and Russell, R. 2001. Read Copy Update. *Ottawa Linux Symposium*. Linux Symposium.

McKenney, P. E., Sarma, D., Arcangeli, A., Kleen, A., Krieger, O., and Russell, R. 2002. Read copy update. In *Proceedings of the Ottawa Linux Symposium (OLS)*. 338–367.

McKenney, P. E. and Slingwine, J. 1993. Efficient kernel memory allocation on shared-memory multiprocessor. In *USENIX Technical Conference Proceedings*. USENIX, Berkeley, CA, 295–305.

McKenney, P. E. and Slingwine, J. 1998. Read-Copy Update: Using exeuction history to solve concurrency problems. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Y. Pan, S. G. Akl, and K. Li, Eds. IASTED/ACTA Press, Las Vegas, NV.

Mukherjee, B. C. and Schwan, K. 1993. Improving performance by use of adaptive objects: experimentation with a configurable multiprocessor thread package. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*. IEEE, 59–66.

Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S. 1980. Medusa: An experiment in distributed operating system structure. *Comm. ACM 23*, 2 (Feb.), 92–104.

Papamarcos, M. S. and Patel, J. H. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*. ACM Press, New York, NY, 348–354.

Peacock, J. K., Saxena, S., Thomas, D., Yang, F., and Yu, W. 1992. Experiences from multithreading System V Release 4. In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*. USENIX, Berkeley, CA, 77–92.

Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weise, J. 1985. The IBM research parallel processor prototype (RP3): Introduction. In *Proceedings of the International Conference on Parallel Processing*.

Pike, R. 1998. Personal communication.

Presotto, D. L. 1990. Multiprocessor streams for Plan 9. In *Proceedings of the Summer UKUUG Conference* London, OR 11–19.

Rashid, R. 1986. From RIG to accent to mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*. 1128–1137. Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Rashid, R., Tevanian, Jr., A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. J., and Chew, J. 1988. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. Comput. 37 8*, 896–908.

Rosenblum, M., Bugnion, E., Herrod, S. A., Witchel, E., and Gupta, A. 1995. The impact of architectural trends on operating system performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*. ACM Press, 285–298.

Russell, C. H. and Waterman, P. J. 1987. Variations on UNIX for parallel-programming computers. *Comm. ACM 30*, 12 (Dec.), 1048–1055.

SCHWAN, K. AND BO, W. 1990. Topologies: distributed objects on multicomputers. *ACM Trans. Comput. Syst.* 8, 2, 111–157.

SCOTT, M. L., LEBLANC, T. J., AND MARSH, B. D. 1988. Design rationale for Psyche, a general-purpose multiprocessor operating system. In *Proceedings of the International Conference on Parallel Processing*. Pennsylvania State University Press, St. Charles, IL, 255. (Also published in the Univ. of Rochester 1988-89 CS and Computer Engineering Research Review.)

SHAPIRO, M., GOURBANT, Y., HABERT, S., MOSSERI, L., RUFFIN, M., AND VALOT, C. 1989. SOS: An object-oriented operating system—assessment and perspectives. *Comput. Syst. 2*, 4, 287–337.

SMITH, B. 1994. The quest for general-purpose parallel computing. www.cray.com/products/systems/mta/psdocs/nsf-agenda.pdf.

SOULES, C. A. N., APPAVOO, J., HUI, K., WISNIEWSKI, R. W., DA SILVA, D., GANGER, G. R., KRIEGER, O., STUMM, M., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., AND XENIDIS, J. 2003. System support for online reconfiguration. In *USENIX Conference Proceedings*. San Antonio, TX, 141–154.

SPEC.ORG. 1996. SPEC SDM suite. http://www.spec.org/osg/sdm91/.

SQUILLANTE, M. S. AND LAZOWSKA, E. D. 1993. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parall. Distrib. Syst. 4*, 2 (Feb.), 131–143.

STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., AND SCOTT, M. 1997. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*.

TALBOT, J. 1995. Turning the AIX operating system into an MP-capable OS. In *Proceedings of the USENIX Technical Conference*.

TORRELLAS, J., GUPTA, A., AND HENNESSY, J. L. 1992. 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 Systems*. Boston, MA, 162–174.

TORVALDS, L. 2002. Posting to linux-kernel mailing list: Summary of changes from v2.5.42 to v2.5.43. http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2.

UNRAU, R. C., KRIEGER, O., GAMSA, B., AND STUMM, M. 1995. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *J. Supercomput. 9*, 1–2, 105–134.

VAN STEEN, M., HOMBURG, P., AND TANENBAUM, A. S. 1997. The architectural design of Globe: A wide-area distributed sytem. Tech. rep. IR-442, Vrige Universiteit, De Boelelann 1105, 1081 HV Amsterdam, The Netherlands.

VASWANI, R. AND ZAHORJAN, J. 1991. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of 13th ACM Symposium on Operating Systems Principles (SOSP-13)*. ACM Press, 26–40.

VERGHESE, B., DEVINE, S., GUPTA, A., AND ROSEMBLUM, M. 1996. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, MA 279–289.

VRANESIC, Z. G., STUMM, M., LEWIS, D. M., AND WHITE, R. 1991. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Comput. 24*, 1 (Jan.), 72–80.

WULF, W., LEVIN, R., AND PIERSON, C. 1975. Overview of the Hydra operating system development. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP-5)*. ACM Press, 122–131.

XIA, C. AND TORRELLAS, J. 1996. Improving the Performance of the data memory hierarchy for multiprocessor operating systems. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture* (*HPCA-2*).

YOUNG, M., TEVANIAN, JR., A., RASHID, R., GOLUB, D., EPPINGER, J. L., CHEW, J., BOLOSKY, W. J., BLACK, D., AND BARON, R. 1987. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP-11)*. ACM Press, 63–76.

Received December 2005; revised August 2006; January 2007; accepted January 2007