Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores

David Wentzlaff and Anant Agarwal

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{wentzlaf, agarwal}@csail.mit.edu

Abstract

The next decade will afford us computer chips with 100's to 1,000's of cores on a single piece of silicon. Contemporary operating systems have been designed to operate on a single core or small number of cores and hence are not well suited to manage and provide operating system services at such large scale. If multicore trends continue, the number of cores that an operating system will be managing will continue to double every 18 months. The traditional evolutionary approach of redesigning OS subsystems when there is insufficient parallelism will cease to work because the rate of increasing parallelism will far outpace the rate at which OS designers will be capable of redesigning subsystems. The fundamental design of operating systems and operating system data structures must be rethought to put scalability as the prime design constraint. This work begins by documenting the scalability problems of contemporary operating systems. These studies are used to motivate the design of a factored operating system (fos). *fos is a new operating system targeting manycore systems with scalability as the primary design constraint, where space sharing replaces time sharing to increase scalability.* We describe fos, which is built in a message passing manner, out of a collection of Internet inspired services. Each operating system service is factored into a set of communicating servers which in aggregate implement a system service. These servers are designed much in the way that distributed Internet services are designed, but instead of providing high level Internet services, these servers provide traditional kernel services and replace traditional kernel data structures in a factored, spatially distributed manner. fos replaces time sharing with space sharing. In other words, fos's servers are bound to distinct processing cores and by doing so do not fight with end user applications for implicit resources such as TLBs and caches. We describe how fos's design is well suited to attack the scalability challenge of future multicores and discuss how traditional application-operating systems interfaces can be redesigned to improve scalability.

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

General Terms Operating System Design, Multicore Computers

Keywords Multicore Operating Systems, Factored Operating System

To appear in ACM SIGOPS Operating System Review: Special Issue on the Interaction among the OS, Compilers, and Multicore Processors. April 2009.

1. Introduction

The number of processor cores which fit onto a single chip microprocessor is rapidly increasing. Within ten years, a single microprocessor will contain 100's - 1,000's cores. Current operating systems were designed for single processor or small number of processor systems and were not designed to manage such scale of computational resources. Unlike the past, where new hardware generations brought higher clock frequency, larger caches, and more single stream speculation, all of which are not huge changes to fundamental system organization, the multicore revolution promises drastic changes in fundamental system architecture, primarily in the fact that the number of general-purpose schedulable processing elements is drastically increasing. The way that an operating system manages 1,000 processors is so fundamentally different than the manner in which it manages one or two processors that the entire design of an operating system must be rethought. This work investigates why simply scaling up traditional symmetric multiprocessor operating systems is not sufficient to attack this problem and proposes how to build a factored operating system (fos) which embraces the 1,000 core multicore chip opportunity.

The growing ubiquity of multicore processors is being driven by several factors. If single stream microprocessor performance were to continue increasing exponentially, there would be little need to contemplate parallelization of our computing systems. Unfortunately, single stream performance of microprocessors has fallen off the exponential trend due to the inability to detect and exploit parallelism in sequential codes, the inability to further pipeline sequential processors, the inability to raise clock frequencies due to power constraints, and the design complexity of high-performance single stream microprocessors[4]. While single stream performance may not be significantly increasing in the future, the opportunity provided by semiconductor process scaling is continuing for the foreseeable future. The ITRS road-map[1] and the continuation of Moore's Law[16] forecast exponential increases in the number of transistors on a single microprocessor chip for at least another decade. In order to turn these exponentially increasing transistor resources into exponentially increasing performance, microprocessor manufacturers have turned to integrating multiple processors onto a single die. Current examples of this include Intel and AMD's Quad-core offerings, TI's TMS320C6474 triple core offering, Freescale's 6-core MSC8156, Tilera's 64-core processor[25], and an 80-core Intel prototype processor[22]. Road-maps by all major microprocessor manufacturers suggest that the trend of integrating more cores onto a single microprocessor will continue. Extrapolating the

doubling of transistor resources every 18-months, and that a 64-core commercial processor was shipped in 2007, in just ten years, we will be able to integrate over 6,000 processor cores on a single microprocessor.

The fact that single stream performance has sizably increased with past generations of microprocessors has shielded operating system developers from qualitative hardware platform changes. Unlike quantitative changes such as larger caches, larger TLBs, higher clock frequency, and more instruction level parallelism, the multicore phenomenon is a qualitative change which drastically changes the playing field for operating system design. The primary challenge of multicore operating system design is one of scalability. Current symmetric multiprocessor (SMP) operating systems have been designed to manage a relatively small number of cores. The number of cores being managed has stayed relatively constant with the vast majority of SMP systems being two processor systems. With multicore chip designs, the number of cores will be expanding at an exponential rate therefore any operating system designed to run on multicores will need to embrace scalability and make it a first order design constraint.

This work investigates the problems with scaling SMP OS's to high core counts. The first problem is that scaling SMP OS's by creating successively finer grain data structure locks is becoming problematic. Unlike small core count systems, where only a small portion of the code may need fine grain locking, in high core count systems, any non-scalable portion of the design will quickly become a performance problem by Ahmdal's law. Also, in order to build an OS which performs well on 100 and 10,000 cores, there may be no optimal lock granularity as finer grain locking allows for better scaling, but introduces potential lock overhead on small core count machines. Last, retrofitting fine grain locking into an SMP OS can be an error prone and challenging prospect.

A second challenge SMP OS's face is that they rely on efficient cache coherence for communications of data structures and locks. It is doubtful that future multicore processors will have efficient full-machine cache coherence as the abstraction of a global shared memory space is inherently a global shared structure. Another challenge for any scalable OS is the need to manage locality. Last, the design of SMP OS's traditionally execute the operating system across the whole machine. While this has good locality benefits for application and OS communications, it requires the cache system on each core of a multicore system to contain the working set of the application and OS.

This work utilizes the Linux 2.6 kernel as a vehicle to investigate scaling of a prototypical SMP OS. We perform scaling studies of the physical page allocation routines to see how varying core count affects the performance of this parallelized code. We find that the physical page allocator does not scale beyond 8 cores under heavy load. We also study the cache performance interference when operating system and application code are executed on the same core.

We use these scalability studies to motivate the design of a factored operating system (fos). fos is a new scalable operating system targeted at 1000+ core systems. The main feature of fos is that it factors an OS into a set of services where each service is built to resemble a distributed Internet server. Each system service is composed of multiple server processes which are spatially distributed across a multicore chip. These servers collaborate and exchange information, and in aggregate provide the overall system service. In fos, each server is allocated to a specific core thereby removing the need to time-multiplex processor cores and simplifying the design of each service server.

fos not only distributes high-level services, but also, distributes services and data-structures typically only found deep in OS kernels such as physical page allocation, scheduling, memory management, naming, and hardware multiplexing. Each system service is constructed out of collaborating servers. The system service servers execute on top of a microkernel. The fos-microkernel is platform dependent, provides protection mechanisms but not protection policy, and implements a fast machine-dependent communication infrastructure.

Many of fos's fundamental system services embrace the distributed Internet paradigm even further by allowing any core to contact any server in a particular system service group. This is similar to how a web client can access any webserver in a load balanced web cluster, but for kernel data structures and services. Also, like a spatially load balanced web cluster, the fos approach exploits locality by distributing servers spatially across a multicore. When an application needs a system service, it only needs to communicate with its local server thereby exploiting locality.

Implementing a kernel as a distributed set of servers has many advantages. First, by breaking away from the SMP OS monolithic kernel approach, the OS level communication is made explicit and exposed thus removing the problem of hunting for poor performing shared memory or lock based code. Second, in fos, the number of servers implementing a particular system service scales with the number of cores being executed on, thus the computing available for OS needs scales with the number of cores in the system. Third, fos does not execute OS code on the same cores which are executing application code. Instead an application messages the particular system service, which then executes the OS code and returns the result. By partitioning where the OS and applications execute, the working set of the OS and the working set of the application do not interfere.

Another impediment to the scalability of modern day operating systems is that the operating system interfaces provided to applications are inherently non-scalable. One way to increase scalability is to provide interfaces which supply information to the user in a best effort manner. For example, by allowing best-effort information, it empowers the user to request information in a high performance manner which may be slightly out of date. Alternatively, the user can select the lower performing, but accurate interface.

fos's design is inspired by microkernels such as Mach [2], L3, and L4 [15] but has significant differences. These differences include:

- fos distributes and parallelizes within a single system service server.
- fos embraces the spatial nature of multicore processors by having a spatially aware placement engine/scheduler.
- Because applications and OS execute on different cores, fos does not need to take an expensive context switch when an application messages a OS server.

Section 4 provides more detail on related research.

We currently have a prototype implementation of the fos-microkernel. It runs on 16 core x86_64 hardware and a QEMU simulation of up to 255 processors. The operating system is under active development and currently consists of a bootloader and a microkernel with a messaging layer. We now developing system service servers.

This paper is organized as follows. Section 2 identifies the scalability problems with contemporary SMP operating systems. Section 3 describes the design of fos and how its design attacks scalability problems. Section 4 describes related work and finally we conclude

2. Scalability Problems of Contemporary Operating Systems

This section investigates three main scalability problems with contemporary OS design, locks, locality aliasing and reliance on shared memory. Case studies are utilized to illustrate how each of these problems appears in a contemporary OS, Linux, on modern multicore x86_64 hardware. The results of these studies are utilized to make recommendations for future operating systems.

2.1 Locks

Contemporary operating systems which execute on multi-processor systems have evolved from uni-processor operating systems. The most simplistic form of this evolution was the addition of a single big kernel lock which prevents multiple threads from simultaneously entering the kernel. Allowing only one thread to execute in the kernel at a time greatly simplifies the extension of a uni-processor operating system to multiple processors. By allowing only one thread in the kernel at a time, the invariant that all kernel data structures will be accessed by only one thread is maintained. Unfortunately, one large kernel lock, by definition, limits the concurrency achievable within an OS kernel and hence the scalability. The traditional manner to further scale operating system performance has been to successively create finer-grain locks thus reducing the probability that more than one thread is concurrently accessing locked data. This method attempts to increase the concurrency available in the kernel.

Adding locks into an operating system is time consuming and error prone. Adding locks can be error prone for several reasons. First, when trying to implement a fine grain lock where coarse grain locking previously existed, it is common to forget that a piece of data needs to be protected by a lock. Many times this is caused by simply not understanding the relationships between data and locks, as most programming languages, especially those commonly used to write operating systems, do not have a formal way to express lock and protected data relationships.

The second manner in which locks are error prone is that locks can introduce circular dependencies and hence cause deadlocks to occur. Many operating systems introduce lock acquisition hierarchies to guarantee that a circular lock dependence can never occur, but this introduces significant complexity for the OS programmer. An unfortunate downside of lock induced deadlocks is that they can occur in very rare circumstances which can be difficult to exercise in normal testing.

When the lock granularity needs to be adjusted it is usually not the case that simply adjusting the lock granularity is enough. For code which has already been parallelized, it is typically difficult to make code finer grain locked in a vacuum. Instead, it is typical for entire sub-systems of the operating system to be redesigned when lock granularity needs to be adjusted.

In previous multiprocessor systems, the speed at which parallelism increased was slow and sub-system redesign could be tackled. In sharp contrast, future multicore processors will follow an exponential growth rate in the number of cores. The effect of this is that each new generation of chip will require the granularity of a lock to be halved in order to maintain performance parity. Thus this lock granularity change may require operating system sub-systems to be redesigned with each new chip generation. Unfortunately for the operating system programmer, it is very difficult to redesign sub-systems with this speed as programmer productivity is not scaling with number of transistors. Hence we believe that traditional lock based operating systems need to be rethought in light of the multicore era.

Whenever discussing lock granularity, the question arises, what is the correct lock granularity? If lock granularity is chosen to be too coarse, the scalability on highly parallel systems may be poor. But, if the lock granularity is too fine, the overhead of locking and unlocking too often can cause inefficiencies on low core-count systems. Future operating systems will have to directly attack finding the correct lock granularity as they will have to span multiple generations of computer chips which will vary by at least an order of magnitude with respect to core count. Also, the difference in core count between the high end processor and low end processor of the same generation may be at least an order of magnitude in the 1000+ core era, thus even within a processor family, the OS designer may not be able to choose an appropriate lock granularity.

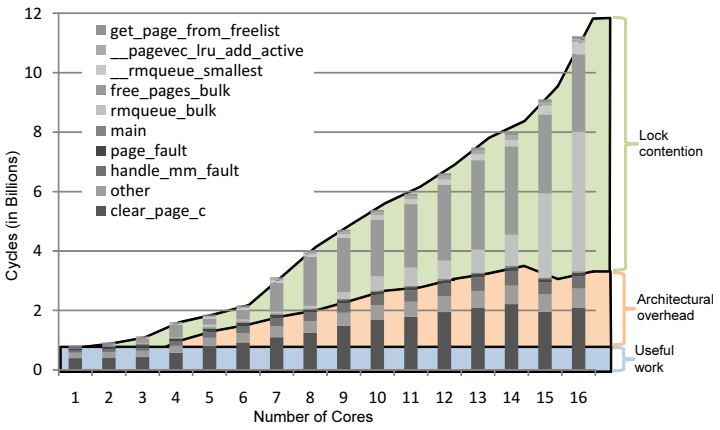

Figure 1. Physical memory allocation performance sorted by function. As more cores are added more processing time is spent contending for locks.

2.1.1 Case Study: Physical Page Allocator

In order to investigate how locks scale in a contemporary operating system, we investigated the scaling aspects of the physical page allocation routines of Linux. The Linux 2.6.24.7 kernel was utilized on a 16 core Intel quad-socket quad-core system. The test system is a Dell PowerEdge R900 outfitted with four Intel Xeon E7340 CPUs running at 2.40GHz and 16GB of RAM.

The test program attempts to allocate memory as quickly as is possible on each core. This is accomplished by allocating a gigabyte of data and then writing to the first byte of every page as quickly as is possible. By touching the first byte in every page, the operating system is forced to demand allocate the memory. The number of cores was varied from 1 to 16 cores. Precision timers and oprofile were utilized to determine the runtime and to profile the executing code. Figure 1 shows the results of this experiment. The bars show the time taken to complete the test per core. Note that a fixed amount of work is done per core, thus perfect scaling would be bars all the same height.

By inspecting the graph, several lessons can be learned. First, as the number of cores increases, the lock contention begins to dominate the execution time. Beyond eight processors, the addition of more processors actually slows down the computation and the system begins to exhibit fold-back. We highlight architectural overhead as time taken due to the hardware not scaling as more cores are added. The architectural overhead is believed to be caused by contention in the hardware memory system.

For this benchmark, the Linux kernel already utilizes relatively fine-grain locks. Each core has a list of free pages and a per-core lock on the free list. There are multiple memory zones each with independent lock sets. The Linux kernel re-balances the free lists in bulk to minimize re-balancing time. Even with all of these optimizations, the top level re-balancing lock ends up being the

scalability problem. This code is already quite fine-grain locked thus to make it finer grain locked, some algorithmic rethinking is needed. While it is not realistic for all of the cores in a 16 core system to allocate memory as quickly as this test program does, it is realistic that in a 1000+ core system, 16 out of the 1000 cores would need to allocate a page at the same time thus causing traffic similar to this test program.

2.2 OS-Application and OS-OS Locality Aliasing

Operating systems have large instruction and data working sets. Traditional operating systems time multiplex computation resources. By executing operating system code and application code on the same physical core, implicitly shared resources such as caches and TLBs have to accommodate the shared working set of both the application and the operating system code and data. This reduces the hit rates in these cache structures versus executing the operating system and application on separate cores. By reducing cache hit rates, the single stream performance of the program will be reduced. Reduced hit rate is exacerbated by the fact that many-core architectures typically contain smaller per-core caches than past uniprocessors.

Single stream performance is at a premium with the advent of multicore processors as increasing single stream performance by other means may be exceedingly difficult. It is also likely that some of the working set will be so disjoint that the application and operating system can fight for resources causing anti-locality collisions in the cache. Anti-locality cache collisions are when two different sets of instructions pull data into the cache at the same index hence causing the different instruction streams to destroy temporal locality for data at a conflicting index in the cache. Current operating systems also execute different portions of the OS with wildly different code and data on one physical core. By doing this, intra-OS cache thrash can be accentuated versus executing different logical portions of the OS on different physical cores.

Cache interference also hampers embedded operating systems which offer quality of service (QOS) or real-time guarantees. The variability introduced by OS-application cache interference has caused many embedded applications to eliminate usage of an operating system and elect to use a more bare metal approach.

2.2.1 Case Study: Cache Interference

In order to evaluate the cache system performance degradation due to executing the operating system and application code on the same core, we created a cache tool which allows us to differentiate operating system from application memory references. The tool is based off of the x86_64 version of QEMU, and captures memory references differentiated by protection level. We constructed a cache model simulator to determine the cache miss rates due to the operating system, the application, and the interference misses caused by the operating system and application contending for cache space. This was accomplished by simulating a unified cache, an OS only cache, and an application only cache for differing cache sizes and configurations.

For our workload, we used Apache2 executing on full stack Linux 2.6.18.8 under a Debian 4 distribution. In this test case, Apache2 is serving static webpages which are being accessed over the network by Apache Bench (ab) simulating ten concurrent users. Figure 2 and 3 show the results for this experiment using a direct mapped and two-way set associative cache respectively.

Studying these results, it can be seen that for small cache sizes, the miss rates for the operating system far surpass the miss rates for the application. Second, the miss rate due to cache interference is sizable. This interference can be removed completely when the operating system and application is executed in different cores. Also, by splitting the application away from the operating system,

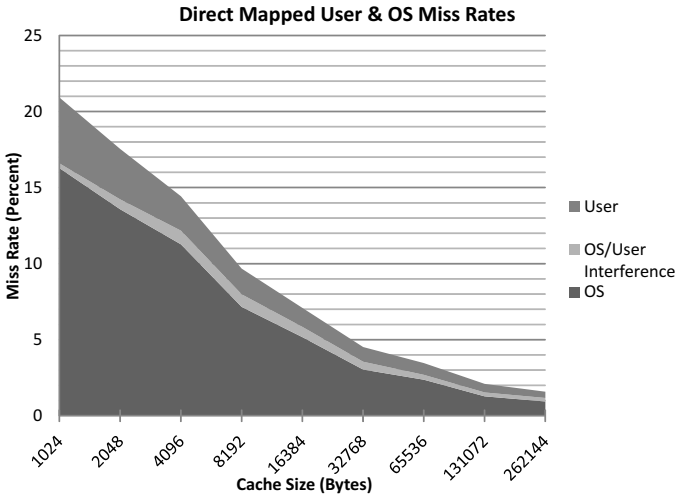

Figure 2. Cache miss rates for Apache2 running on Linux 2.6.18.8. User, OS/user interference, and OS miss rate for a direct mapped cache.

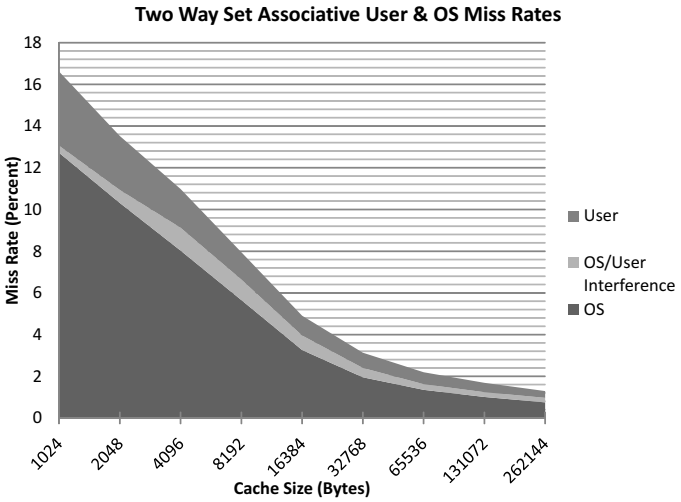

Figure 3. Cache miss rates for Apache2 running on Linux 2.6.18.8. User, OS/user interference, and OS miss rate for a two-way set-associative cache.

it allows the easy utilization of more parallel caches, a plentiful resource on multicore processors, while single stream performance is at a premium. Last, when examining large cache sizes, the percentage of interference misses grows relative to the total number of misses. This indicates that for level-2 and level-3 caches, intermixing cache accesses can be quite quite damaging to performance. This result re-affirms the results found in [3], but with a modern application and operating system.

2.3 Reliance on Shared Memory

Contemporary operating systems rely on shared memory for communication. Largely this is because shared memory is the only means by which a desktop hardware architecture allows core-to-core communication. The abstraction of a flat, global, address space is convenient for the programmer to utilize as addresses can be passed across the machine and any core is capable of accessing the data. It is also relatively easy to extend a single threaded oper-

ating system into a multi-threaded kernel by using a single global address space. Unfortunately, the usage of a single global shared memory is an inherently global construct. This global abstraction makes it challenging for a shared memory system to scale to large core count.

Many current embedded multicore processors do not support a shared memory abstraction. Instead cores are connected by ad-hoc communication FIFOs, explicit communication networks, or by asymmetric shared memory. Current day embedded multicores are pioneers in the multicore field which future multicore processors will extend. Because contemporary operating systems rely on shared memory for communication, it is not possible to execute them on current and future embedded multicores which lack full shared memory support. In order to have the widest applicability, future multicore operating systems should not be reliant on a shared memory abstraction.

It is also unclear whether cache coherent shared memory will scale to large core counts. The most promising hardware shared memory technique with respect to scalability has been directory based cache coherence. Hardware directory based cache coherence has found difficulties providing high performance cache coherent shared memory above about 100 cores. The alternative is to use message passing which is a more explicit point-to-point communication mechanism.

Besides scalability problems, modern operating system's reliance on shared memory can cause subtle data races. If used incorrectly, global shared memory easily allows the introduction of data races which can be difficult to detect at test time.

2.4 Recommendations

The problems presented in this section lead to a few recommendations in order to improve scalability for future operating systems, namely:

- Avoid the use of hardware locks.
- Separate the operating system execution resources from the application execution resources.
  - Reduces implicit resource sharing (Caches/TLBs).
  - Utilizes the ample thread parallelism provided by multicore processors.
- Avoid global cache coherent shared memory
  - Broadens applicability to architectures which don't support shared memory.
  - Shared memory hardware may not scale to large core count.

3. Design of a Factored Operating System

In order to create an operating system to tackle the 1,000+ core era, we propose designing a factored operating system (fos). fos is an operating system which takes scalability as *the* first order design constraint. Unlike most previous operating systems where a subsystem scales up to a given point and beyond that point, the subsystem needs to be redesigned, fos ventures to develop techniques to build operating system services which scale, up and down, across a large range ($>$ 3 orders of decimal magnitude) of core count.

3.1 Design Principles

In order to achieve the goal of scaling over multiple orders of magnitude in core count, fos utilizes the following design principles:

- Space multiplexing replaces time multiplexing.
  - Scheduling becomes a layout problem not a time multiplexing problem.
  - OS runs on distinct cores from applications.
  - Working sets are spatially partitioned; OS does not interfere with applications cache.
- OS is factored into function specific services.
  - Each OS service is distributed into spatially distributed servers.
  - Servers collaborate and communicate only via message passing.
  - Servers are bound to a core.
  - Applications communicate with servers via message passing.
  - Servers leverage ideas (caching, replication, spatial distribution, lazy update) from Internet servers.

In the near future, we believe that the number of cores on a single chip will be on the order of the number of active threads in a system. When this occurs, the reliance on temporal multiplexing of resources will be removed, thus fos replaces traditional time multiplexing with space multiplexing. By scheduling resources spatially, traditional scheduling problems are transformed into layout and partitioning problems. We believe that balancing may still occur, but in fos, balancing will occur significantly less often than in an operating system which temporally multiplexes resources. Determining the correct placement of processes for good performance will be the challenge that future operating systems will face and one that fos tackles.

Spatial multiplexing is taken further as fos is factored into function specific services. Each system service is then distributed into a fleet of cooperating servers. All of the different function specific servers collaborate to provide a service such as an a file system interface. Each server is bound to a particular processor core and communicates with other services in the same service fleet via messaging only. When an application needs to access a service provided by the operating system, the application messages the closest core providing that service. The closest core providing a particular service is found by querying a name server.

In order to build a scalable operating system, the fos server is inspired by Internet servers. But, instead of serving web pages, fos servers manage traditional kernel operations and data structures. fos servers leverage ideas such as extensive caching, replication, spatial distribution, and lazy update, which have allowed Internet services to scale up to millions of users. Writing fos servers can be more challenging than writing traditional shared memory operating system subsystems. We believe that the shared nothing, message passing only, approach will encourage OS developers to think very carefully about exactly what data is being shared, which will lead to more scalability.

3.2 Structure of fos

A factored operating system environment is composed of three main components. A thin microkernel, a set of servers which together provide system services which we call the OS layer, and applications which utilize these services. The lowest level of software management comes from the microkernel. A portion of the microkernel executes on each processor core. The microkernel controls access to resources (protection), provides a communication API and code spawning API to applications and system service servers, and maintains a name cache used internally to determine the location (physical core number) of the destination of messages. Applications and system servers execute on separate cores on top of the microkernel and execute on the same core resources as the microkernel as shown in Figure 4.

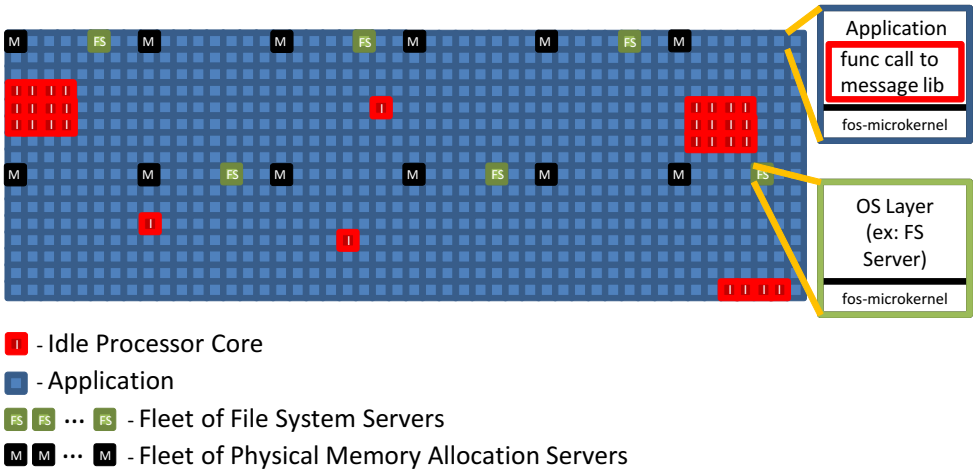

Figure 4. OS and application clients executing on the fos-microkernel

fos is a full featured operating system which provides many services to applications such as resource multiplexing, management of system resources such as cores, memory, and input-output devices, abstraction layers such as file-systems and networking, and application communication primitives. In fos, this functionality is provided by the OS layer. The OS layer is composed of fleets of function specific servers. Each operating system function is provided by one or more servers. Each server of the same type is a part of a function specific fleet. Naturally there are differing fleets for different functions. For instance there is a fleet which manages physical memory allocation, a fleet which manages the file system access, and a fleet which manages process scheduling and layout. Each server executes solely on a dedicated processor core. Servers communicate only via the messaging interface provided by the microkernel layer.

In fos, an application executes on one or more cores. Within an application, communication can be achieved via shared memory communication or messaging. While coherent shared memory may be inherently unscalable in the large, in a small application, it can be quite useful. This is why fos provides the ability for applications to have shared memory if the underlying hardware supports it. The OS layer does not internally utilize shared memory, but rather utilizes explicit message based communication. When an application requires OS services, the underlying communication mechanism is via microkernel messaging. While messaging is used as the communication mechanism, a more traditional system call interface is exposed to the application writer. A small translation library is used to turn system calls into messages from application to an OS layer server.

Applications and OS layer servers act as peers. They all run under the fos-microkernel and communicate via the fos-microkernel messaging API. The fos-microkernel does not differentiate between applications and OS layer servers executing under it. The code executing on a single core under the fos-microkernel is called an fos client. Figure 4 has a conceptual model of applications and the OS layer, as implemented by fleets of servers, executing on top of the microkernel. As can be seen from the figure, fos concentrates on spatial allocation of resources over time multiplexing of resources. In high core count environments, the problem of scheduling turns from one of time slicing to one of spatial layout of executing processes.

3.2.1 Microkernel Messaging

The key service provided by the fos-microkernel to microkernel clients is that of a reliable messaging layer. Each fos-microkernel client can allocate a large number of receive mailboxes via which it can receive messages. Clients can attach symbolic names to their receive mailboxes and publish these names such that other clients can find published services. Namespace can be protected to prevent clients from stealing namespace from privileged service names. fos clients send messages to named mailboxes. The fos-microkernel manages the reliable transport and enqueuing of messages into the receiving mailbox. If a receive mailbox is full, the sending client *send* call returns an error.

In addition to transport of messages, the fos-microkernel maintains a cache of name mapping. By using a cache based system, the fos-microkernel can provide fast name lookups and a large namespace at the same time. The fos-microkernel delegates destination look-up to the name server fleet (running in the OS layer) which maintains the canonical mailbox name to physical core and mailbox directory. Utilizing a name server allows for redirection of mailboxes if a client changes physical location. The name server also provides a one-to-many mapping function allowing multiple clients to implement the same service in a server pool manner. This enables a fleet of servers to implement the same function and allow applications to access them via one known name. For one-to-many mappings, the name server can choose a particular server instance based off of physical proximity or load balancing. Also, the name server can be used to provide fault resilience as broken servers can be steered away from when a fault is detected. fos is primarily focused on scalability and not fault tolerance. To that end, on-chip communications are considered to be reliable, but if a fault is detected corrective actions are taken.

Figure 5 diagrams an example file system access. 1: An application calls *read* which calls the message proxy library. 2: The message proxy library constructs a message to the file system service. 3: The message proxy library calls the fos-microkernel to send the message. 4: The fos-microkernel looks up the physical destination in name cache. 5: The fos-microkernels transport the message to the destination via on-chip networks or shared memory. 6: The receive microkernel deposits message in the file system server's request mailbox. 7: The file system server processes the request. 8: The file system server returns data to the message proxy library receive mailbox via a message which follows a similar return path. 9:

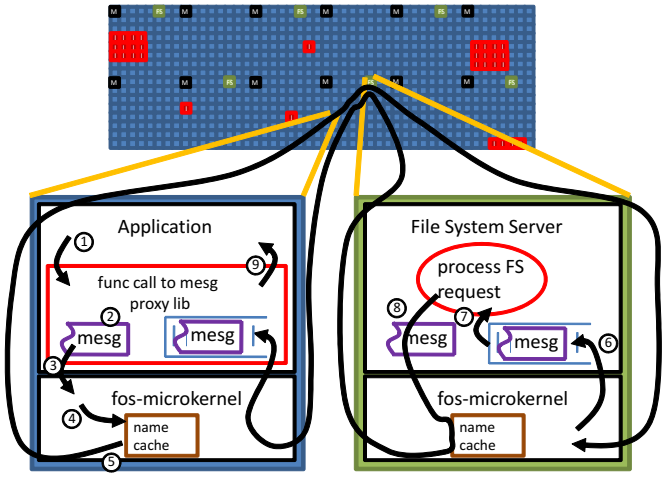

Figure 5. Message walkthrough of an example application file system access.

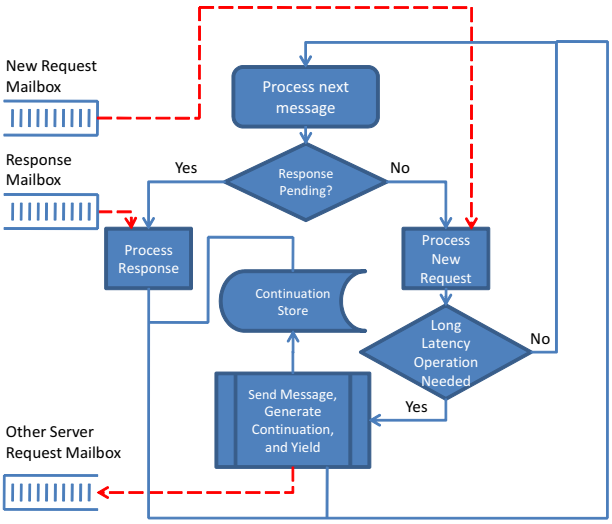

Figure 6. The main runloop for a server.

The message proxy library unpackages the response message and returns data to the application.

3.2.2 Microkernel to OS Layer Delegation

The fos-microkernel is designed to delegate functionality to the OS layer in several situations. One example is the naming service maintained in the OS layer, but the fos-microkernel needs to access this information to route messages. While this may seem to cause cyclic dependencies, the fos-microkernel and delegated clients have been designed with this in mind. The fos-microkernel and delegated client communicate via microkernel messaging. In order to prevent dependency cycles, the fos-microkernel knows the physical location of the delegated to client tasks. Also, delegated to client tasks are not dependent on microkernel services which they ultimately provide. Example delegated services are the name server service and the privilege service.

The privilege service is implemented as a fleet of servers in the OS layer. The fos-microkernel requests privilege information from the delegated to privilege manager servers and caches the information inside of the microkernel in a read only manner. Occasionally privileges change and the privilege manager messages the microkernel notifying the microkernel to invalidate the appropriate stale privilege information. In order for the privilege manager to run as a fos-microkernel client, the fos-microkernel affords privilege manager servers static privileges, so that a privilege fixed point can be reached.

3.2.3 Structure of a Server

fos's servers are inspired by Internet servers. The typical server is designed to process an inbound queue of requests and is transaction-oriented. A transaction consists of a request sent to a server, the server performing some action, and a reply being sent. Most fos servers are designed to use stateless protocols like many Internet services. This means that each request encodes all of the needed data to complete a transaction and the server itself does not need to store data for multiple transactions in sequence. By structuring servers as transaction-oriented stateless processes, server design is much simplified. Also, scalability and robustness is improved as requests can be routed by the name server to differing servers in the same server fleet.

Programming difficulty of a typical server is also reduced because each server processes a transaction to completion without the possibility of interruption. Thus local locking is not required to prevent multiple server threads from attempting to concurrently update memory. Some transactions may require a long latency operation to occur, such as accessing I/O or messaging another server. When a long latency operation does occur, a server constructs a continuation for the current transaction, which is stored locally. The continuation is restarted when a response from the long latency operation is received.

Servers are structured to process two inbound mailboxes. One for new requests and one for responses. The server prioritizes responses over requests. Servers do not preempt transactions, but rather use a cooperative model. Figure 6 shows the typical control flow of a server. Servers are designed to acquire all resources needed to complete a transaction before a transaction creates a continuation and yields to the next transaction. By doing so, servers can be designed without local locking.

3.3 Comparison to Traditional OS's

In this section we evaluate the design of fos to see how it tackles the scalability challenges set forth in Section 2. The first challenge that traditional operating systems face is their utilization of locks to protect access to data which is shared amongst multiple threads.

fos approaches the lock problem with a multipronged approach. First, fos is an inherently message passing operating system, therefore there are no shared memory locks between cores. Second, fos servers are constructed in a non-preemptive manner. Only one thread is ever executing on a server and the server chooses where, when, and if it should yield when a long latency operation occurs. The server writer chooses the yield locations, thus if the data is consistent, no locks are needed. In fos, if a lock is absolutely necessary, hardware locking instructions are not used as servers are not preempted.

In fos, servers can be used as lock managers. While this is possible, the message passing nature of fos discourages this usage model by making programming in this manner difficult. If lock managers are used, the OS designer is able to add more lock management resources explicitly rather than relying on the underlying shared memory system and exclusive operations provided by the hardware.

Applications and the operating system have largely different working sets. To address this problem and avoid implicitly sharing hardware structures such as caches and translation lookaside buffers (TLBs), fos executes the operating system on separate pro-

cessing cores from the application. Also, different portions of the operating system are factored apart such that different operating system working sets do not interfere in implicitly accessed data structures.

The third identified problem with traditional OS's is their dependence on shared memory. Being dependent on hardware shared memory limits the applicability of traditional OS's to multicore hardware architectures which are patterned after symmetric multiprocessors (SMPs). Many current multicore processors, future multicore processors, and embedded multicores do not support shared memory abstractions. fos breaks traditional OS's dependence on shared memory by solely using message passing for internal OS layer communication. By breaking this dependence, fos has a wider applicability than if it was constrained by using shared memory for internal kernel communication.

Shared memory may also limit scalability as currently there is no known scalable cache-coherent shared memory protocol. Also, it is easy to construct subtly incorrect shared memory programs. To combat these problems, fos does not use shared memory inside of the OS layer. We believe that by making data movement explicit the ultimate goal of scalability can be easier to achieve.

One challenge presented by utilizing only message passing is that it decouples the communication and storage of data. Shared memory abstractions allow for shared data to be written to a region of memory which is later read by an unknown reader, while in message passing, the destination of any piece of data must be known when a message is sent. One way that fos combats this problem is that it uses transactions to pull data when needed from the originator of the shared data.

The messaging layer provided by the fos-microkernel is designed to alleviate some of the pains of message passing. First, send routines and receive routines do not need to be synchronized like in the L4 [15] operating system. fos provides large buffers which live in the receiving client's address space. The use of large input mailboxes decouples send and receive synchronization.

3.4 Challenges

fos executes operating system code on disparate processing cores from the application code. This introduces core-to-core communication latency to application-operating system communication. Conversely, when the operating system and application execute on the same core, a context switch is needed when the application and operating system communicate. Typical context switch times are in the 100 cycle to 300 cycle range on modern processors. On multicore architectures with exposed on-chip messaging networks such as the Raw [24, 21] and Tilera [25] processors, the cost of core-to-core dynamic messages is on the order of 15 - 45 cycles depending on distance. fos explores whether separating operating system code away from application code is a good performance tradeoff. We believe that the low communications cost of local core-to-core communication motivates separating operating system code from application code.

One open question for fos is whether structures which are traditionally shared between multiple OS services can be accessed with good performance in an environment such as fos. An example of this type of sharing is the virtual memory system and the file system's buffer cache. Buffer caches are typically globally distributed between all cores and accessed via shared memory. In fos, the servers which provide the file system and the virtual memory management do not share memory with each other. fos replaces wide sharing of data with techniques utilized by distributed services. One example of this replacement is that fos utilizes replication and aggressive caching to replace a widely shared buffer cache. The cost of utilizing implicitly shared memory on future multicores will affect the tradeoff of whether a hardware solution or the software solution explored by fos proves more scalable.

While fos's servers are designed without memory based locks within a server and without locks between servers, synchronization points are still needed to guarantee exclusive access to certain structures when atomicity is required. In fos, a notional lock can be managed by a server. In order to acquire the lock, a message is sent to the owner of the lock which can dole out the lock. One challenge of using lock servers is that the cost of software-based lock management may dwarf the performance wins of removing traditional locks from the operating system. We believe that the cost of using notional locks will affect the design of fos's servers such that notional locks will only be used in rare circumstances.

An alternative to using lock servers is to use a core as a object server or transaction server. Other servers would work in parallel doing the bulk of the computation, only communicating with the transaction server when an operation is complete. By serializing transactions against a transaction server, fos can take advantage of optimistic concurrency.

Another open question is whether it will be possible to write all the needed multi-sequence operations without local locks in a server. For instance if a server needs to execute a sequence of communications with other servers, can all of the servers that fos will contain be designed such that no local resources need to be locked. Due to the single threaded nature of fos's servers, each thread implicitly takes out a global lock on all of the memory owned by one server, but when executing a sequence of operations, it is not certain that all cases can be handled.

3.5 Implementation

fos is currently booting on 16 core x86_64 hardware and a QEMU simulation of up to 255 processors. The operating system is under active development and currently consists of a bootloader, microkernel, messaging layer with capability checking, and name server. We are beginning to implement OS Layer servers.

4. Related Work

There are several classes of systems which have similarities to fos proposed here. These can be roughly grouped into three categories: traditional microkernels, distributed operating systems, and distributed Internet-scale servers.

A microkernel is a minimal operating system kernel which typically provides no high-level operating system services in the microkernel, but rather provides mechanisms such as low level memory management and inter-thread communication which can be utilized to construct high-level operating system services. High-level operating system services are typically constructed inside of servers which utilize the microkernel's provided mechanisms. Mach [2] is an example of an early microkernel. In order to address performance problems, portions of servers were slowly integrated into the Mach microkernel to minimize microkernel/server context switching overhead. This led to the Mach microkernel being larger than the absolute minimum. The L4 [15] kernel is another example of a microkernel which attempts to optimize away some of the inefficiencies found in Mach and focuses heavily on performance.

fos is designed as a microkernel and extends microkernel design. It is differentiated from previous microkernels in that instead of simply exploiting parallelism between servers which provide different functions, this work seeks to distribute and parallelize within a server for a single high-level function. This work also exploits the spatial-ness of massively multicore processors. This is done by spatially distributing servers which provide a common function. This is in contrast to traditional microkernels which were not spatially aware. By spatially distributing servers which collaboratively provide a high-level function, applications which use a given service

may only need to communicate with the local server providing the function and hence can minimize intra-chip communication. Operating systems built on top of previous microkernels have not tackled the spatial non-uniformity inherent in massively multicore processors. fos embraces the spatial nature of future massively multicore processors and has a scheduler which is not only temporally aware, but also spatially aware.

The cost of communication on fos compared to previous microkernels is reduced because fos does not temporally multiplex operating system servers and applications. Therefore when an application messages an OS server, a context swap does not occur. This is in contrast to previous microkernels which temporally multiplexed resources, causing every communication to require a costly context swap. Last, fos, is differentiated from previous microkernels on parallel systems, because the communication costs and sheer number of cores on a massively multicore processor is different than in previous parallel systems, thus the optimizations made and trade-offs are quite different.

The Tornado [11] operating system which has been extended into the K42 [5] operating system is a microkernel operating system and is one of the more aggressive attempts at constructing scalable microkernels. They are differentiated from fos in that they are designed to be run on SMP and NUMA shared memory machines instead of single-chip massively multicore machines. Tornado and K42 also suppose future architectures which support efficient hardware shared memory. fos does not require architectures to support intra-machine shared memory. Also, the scalability claims [6] of K42 have been focused on machines up to 24 processors which is a modest number of processors when compared to the target of 1000+ processors which fos is being designed for.

The Hive [9] operating system utilizes a multicellular kernel architecture. This means that a multiprocessor is segmented into cells which each contain a set of processors. Inside of a cell, the operating system manages the resources inside of the cell like a traditional OS. Between cells the operating system shares resources by having the different cells message and allowing safe memory reads. Hive OS focused heavily on fault containment and less on high scalability than fos does. Also, the Hive results are for scalability up to 4 processors. In contrast to fos, Hive utilizes shared memory between cells as a manner to communicate.

Another approach to building scalable operating systems is the approach taken by Disco [8] and Cellular Disco [13]. Disco and Cellular Disco run off the shelf operating systems in multiple virtual machines executing on multiprocessor systems. By dividing a multiprocessor into multiple virtual machines with fewer processors, Disco and Cellular Disco can leverage the design of pre-existing operating systems. They also leverage the level of scalability already designed into pre-existing operating systems. Disco and Cellular Disco also allow for sharing between the virtual machines in multiple ways. For instance in Cellular Disco, virtual machines can be thought of as a cluster running on a multiprocessor system. Cellular Disco utilizes cluster services like a shared network file system and network time servers to present a closer approximation of a single system image. Various techniques are used in these projects to allow for sharing between VMs. For instance memory can be shared between VMs so replicated pages can point at the same page in physical memory. Cellular Disco segments a multiprocessor into cells and allows for borrowing of resources, such as memory between cells. Cellular Disco also provides fast communication mechanisms which break the virtual machine abstraction to allow two client operating systems to communicate faster than communicating via a virtualized network-like interface. VMWare has adopted many of the ideas from Disco and Cellular Disco to improve VMWare's product offerings. One example is VMCI Sockets [23] which is an optimized communication API which provides fast communication between VMs executing on the same machine.

Disco and Cellular Disco utilize hierarchical shared information sharing to attack the scalability problem much in the same way that fos does. They do so by leveraging conventional SMP operating systems at the base of hierarchy. Disco and Cellular Disco argue leveraging traditional operating systems as an advantage, but this approach likely does not reach the highest level of scalability as a purpose built scalable OS such as fos will. Also, the rigid cell boundaries of Cellular Disco can limit scalability. Last, because at it core these systems are just utilizing multiprocessor systems as a cluster, the qualitative interface of a cluster is restrictive when compared to a single system image. This is especially prominent with large applications which need to be rewritten such that the application is segmented into blocks only as large as the largest virtual machine. In order to create larger systems, an application needs to either be transformed to a distributed network model, or utilize a VM abstraction-layer violating interface which allows memory to be shared between VMs.

More recently, work has been done to investigate operating systems for multicore processors. One example is Corey [7] which focuses on allowing applications to direct how shared memory data is shared between cores.

fos bears much similarity to a distributed operating system, except executing on a single chip. In fact much of the inspiration for this work comes from the ideas developed for distributed operating systems. A distributed operating system is an operating system which executes across multiple computers or workstations connected by a network. Distributed operating systems provide abstractions which allow a single user to utilize resources across multiple networked computers or workstations. The level of integration varies with some distributed operating systems providing a single system image to the user, while others provide only shared process scheduling or a shared file system. Examples of distributed operating systems include Amoeba [20, 19], Sprite [17], and Clouds [10]. These systems were implemented across clusters of workstation computers connected by networking hardware.

While this work takes much inspiration from distributed operating systems, some differences stand out. The prime difference is that the core-to-core communication cost on a single-chip massively multicore processor is orders of magnitude smaller than on distributed systems which utilize Ethernet style hardware to interconnect the nodes. Single-chip massively multicore processors have much smaller core-to-core latency and much higher core-to-core communications bandwidth. A second difference that multicores present relative to clusters of workstations is that on-chip communication is much more reliable than between workstations over commodity network hardware. fos takes advantage of this by approximating on-chip communication as being reliable. This removes the latency of correcting errors and removes the complexity of correcting communication errors. Last, single-chip multicore processors are easier to think of as a single trusted administrative domain than a true distributed system. In many distributed operating systems, much effort is spent determining whether communications are trusted. This problem does not disappear in a single-chip multicore, but the on-chip protection hardware and the fact that the entire system is contained in a single chip simplifies the trust model considerably.

The parallelization of system level services into cooperating servers as proposed by this work has much in common with techniques used by distributed Internet servers. Load balancing is one technique taken from clustered webservers. The name server of fos derives inspiration from the hierarchical caching in the Internet's DNS system. This work hopes to leverage other techniques such as those in peer-to-peer and distributed hash tables such as Bit Tor-

rent, Chord, and Freenet. The file system on fos will be inspired by distributed file systems such as AFS [18], OceanStore [14] and the Google File System [12].

While this work leverages techniques which allow distributed Internet servers to be spatially distributed and provide services at large-scale, there are some differences. First, instead of being applied to serving webpages or otherwise user services, these techniques are applied to services which are internal to an OS kernel. Many of these services have lower latency requirements than are found on the Internet. Second, the on-chip domain is more reliable than the Internet, therefore there are fewer overheads needed to deal with errors or network failures. Last, the communication costs within a chip are orders of magnitude lower than on the Internet.

5. Conclusion

In the next decade, we will have single chips with 100 - 1,000 cores integrated into a single piece of silicon. In this work we chronicled some of the problems with current monolithic operating systems and described these scaling problems. These scaling problems motivate a rethinking of the manner in which operating systems are structured. In order to address these problems we propose a factored operating system (fos) which targets 1000+ core multicore systems and replaces traditional time sharing with space sharing to increase scalability. By structuring an OS as a collection of Internet inspired services we believe that operating systems can be scaled for 1000+ core single-chip systems and beyond allowing us to design and effectively harvest the performance gains of the multicore revolution.

Acknowledgments

This work is funded by DARPA, Quanta Computing, Google, and the NSF. We thank Robert Morris and Frans Kaashoek for feedback on this work. We also thank Charles Gruenwald for help on fos.

References

[1] The international technology roadmap for semiconductors: 2007 edition, 2007. http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, June 1986.

[3] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming workloads. *ACM Transaction on Computer Systems*, 6(4):393–431, Nov. 1988.

[4] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 248–259, June 2000.

[5] J. Appavoo, M. Auslander, M. Burtico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[6] J. Appavoo, M. Auslander, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, J. Xenidis, M. Stumm, B. Gamsa, R. Azimi, R. Fingas, A. Tam, and D. Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. Technical Report RC22863, International Business Machines, July 2003.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. D. Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[8] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 143–156, 1997.

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

[10] P. Dasgupta, R. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. Applebe, J. M. Bernabeu-Auban, P. Hutto, M. Khalidi, and C. J. Wileknloh. The design and implementation of the Clouds distributed operating system. *USENIX Computing Systems Journal*, 3(1):11–46, 1990.

[11] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 87–100, Feb. 1999.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the ACM Symposium on Operating System Principles*, Oct. 2003.

[13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 154–169, 1999.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Nov. 2000.

[15] J. Liedtke. On microkernel construction. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 237–250, Dec. 1995.

[16] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965.

[17] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[18] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–18,20–21, May 1990.

[19] A. S. Tanenbaum, M. F. Kaashoek, R. V. Renesse, and H. E. Bal. The Amoeba distributed operating system-a status report. *Computer Communications*, 14:324–335, July 1991.

[20] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 558–563, May 1986.

[21] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffman, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[22] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 98–99, 589, Feb. 2007.

[23] VMWare, Inc. *VMCI Sockets Programming Guide for VMware Workstation 6.5 and VMware Server 2.0*, 2008. http://www.vmware.com/products/beta/ws/VMCIsockets.pdf.

[24] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, Sept. 1997.

[25] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept. 2007.