Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2008-060 October 8, 2008

The Case for a Factored Operating System (fos)

David Wentzlaff and Anant Agarwal

The Case for a Factored Operating System (fos)

David Wentzlaff and Anant Agarwal
CSAIL, Massachusetts Institute of Technology
Cambridge, MA 02139
{wentzlaf, agarwal}@csail.mit.edu

Abstract

The next decade will afford us computer chips with 1,000 - 10,000 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. Managing 10,000 cores is so fundamentally different from managing two cores that the traditional evolutionary approach of operating system optimization will cease to work. The fundamental design of operating systems and operating system data structures must be rethought. 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 1000+ core multicore systems where space sharing replaces traditional time sharing to increase scalability. *fos is built as a collection of Internet inspired services. Each operating system service is factored into a fleet 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 manage traditional kernel data structures in a factored, spatially distributed manner. The 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. Also, spatial distribution of these OS services facilitates locality as many operations only need to communicate with the nearest server for a given service.*

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 1,000 - 10,000 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 10,000 processors is so fundamentally different than the manner in which it manages two 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 10,000 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[3]. 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[17] 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, Tilera's 64-core processor[26], and an 80-core Intel prototype processor[23]. 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 6000 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 larger caches, larger TLBs, higher clock frequency, and more instruction level parallelism, the multicore phenomenon 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 that they have managed has stayed relatively constant. The number of CPU chips contained within a typical system has remained fixed, 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 OSs 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 node count systems, where only a small portion of the code may need fine grain locking, in high node count systems, any non-scalable portion of the design will quickly become a performance problem. 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 node 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 differing core count effects the performance of this parallelized code. We find that this code does not scale beyond 8 cores under heavy load.

We use these scalability studies to motivate the design of a factored operating system (fos). fos is a new scalable portable operating system targeted at 1000+ core systems. The main feature of fos is that each service that the OS provides is built like 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 node to contact any server in a particular system service group (fleet). 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 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. This work dives into the design of a factored operating system and presents some initial scalability measurements from Linux.

2 Scalability of Contemporary Operating Systems

Computer hardware is always progressing, providing more computing resources year-after-year to the end user. Computer hardware progress has largely been fueled by the ability to produce smaller structures at a lower cost, whether they be transistors on a chip or magnetic regions on a disk. Whenever new hardware is presented to the programmer, software changes are needed to take full advantage of and optimize for the new hardware. Much of the hardware progress over the last 25 years has sought to reduce the qualitative system architecture changes. Instead the bulk of these changes have come in the form of quantitative changes in system architecture such as higher clock frequency, higher ILP, more physical memory, larger caches, larger disks, and higher bandwidth buses. Qualitative changes in system architecture such as the addition of memory management have been significantly less prevalent. From an operating system's perspective, quantitative architecture changes are significantly less disruptive than qualitative changes. Quantitative changes are less disruptive because they typically show up as performance degradation while qualitative differences will show up as loss of functionality.

The advent of 1000+ core microprocessors marks a major qualitative change in computer system design, and one for which operating systems will have to undergo major restructuring. Multi-processor systems have been in use for many years, but have been constrained to low numbers of processors, with the bulk being two processor systems. One of the major differences between desktop and server multiprocessor systems of the past and 1000+ core microprocessors is that in the past the number of processors has stayed constant, in the 2-8 processor range, from generation to generation. Multi-processor systems have been a means to allow users who needed processing power sooner than single stream performance improvement provided a way to utilize more silicon area to gain parallel performance. Mainstream multi-processor systems have largely relied on single stream performance of the individual processors to increase performance instead of having the number of cores exponentially increasing. Unlike past multi-processor systems, the 1000+ core era of multi-core processors enables the number of cores in a system to track the exponential increase in available silicon area. This forces the design of an operating system to treat core count scalability as a first order constraint.

In order to address the exponential increase in cores, 1000+ core operating systems will have to scale in all aspects. Unlike small core count systems where it is acceptable to have small portions of the system software be serialized, when scaled to large core counts, the overheads when applied serially lead to major performance problems. Thus in the 1000+ core case, all portions of the OS must scale well. For example, suppose an 8 core system where 0.1% of the execution time per core is spent in operating system code which executes sequentially with all other processor code. In such a system, less then 0.8% of total execution time is due to OS-level Amdahl's law. But when the same senario is applied to a 1000 core system, an additional 100% of the original execution time is due to OS overheads and the system performance decreases by a factor of two.

In the past, fine-grain locking has been utilized to combat OS scalability problems. Fine-grain locking runs into problems as the number of cores exponentially increases. Assume that the rate at which a user process accesses system services stays constant and the data set is of fixed size. Next, *the probability that a lock will be contended for is proportional to the number of threads in a system. This implies that as multicore systems gain more threads/cores exponentially with time, lock contention grows exponentially worst with time. Thus to achieve performance parity, traditional lock-based operating systems must have every critical section protected by a lock become exponentially smaller every 18 months.* The corollary to this is that the number of locks in a fine-grain locked system, must increase exponentially with time to achieve performance parity. This requirement becomes very difficult as operating system programmer productivity is not even close to being able to identify twice as many lock locations every 18 months. Second, the size of data protected by a single lock must become exponentially smaller every 18 months. Therefore, for contemporary OS code which has already been fine-grain locked on a very small level, it is very probable that the finest grain lock size (a lock per byte?) will be reached very quickly if it has not already reached.

This section investigates three main problems with contemporary OS design, locks, reliance on shared memory, and locality aliasing. 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 studied and lessons learned are utilized to make recommendations for future operating systems.

2.1 Problems

2.1.1 Locks

Contemporary operating systems which execute on multiprocessor 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. Allowing only one thread in the kernel at a time allows the invariant that all kernel data structures will be accessed only by one thread to hold true. 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 a very time consuming and error prone endeavor. Adding locks can be very error prone for several reasons. First, when trying to make make a lock finer grain where course grain locking previously existed, it is very 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 do not have a formal way to express lock and protected data relationships. The second manner in which locks are error prone is when locks can cause circular dependencies and hence deadlocks to occur. This case is quite unfortunate as in many circumstances, these only occur is very rare occurrences and may not be exercised by normal testing. When the lock granularity needs to be adjusted it is usually not the case that simply adjusting the granularity is enough. In many cases, an entire subsystem of the operating system will need to be redesigned in order to take be able to change lock granularity.

2.1.2 Reliance on Shared Memory

Contemporary operating systems rely on shared memory for communication. This is partially because this is the only means by which a typical hardware architecture allows core-to-core communication. A flat global shared memory space is many times convenient to reason about. Also, the easiest extension of a single processor operating system is to allow multiple kernel threads executing in a single global address space.

The downside of relying on shared memory is that shared memory is inherently global and is a challenging to scale up to large scale with good performance. It is doubtful that a hardware solution will be found which provides performent cache coherent shared memory up to 1000's of cores. The alternative is to use message passing which is a more explicit point-to-point communication mechanism.

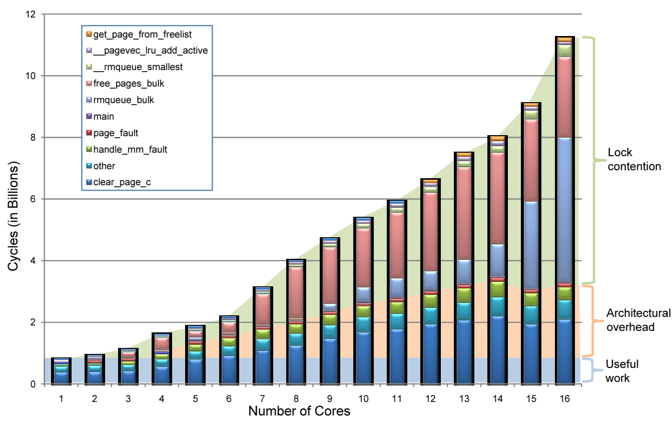

Figure 1. Physical memory allocation performance sorted by function. As more cores are added more processing time is spent contending for locks.

2.1.3 OS-Application and OS-OS Locality Aliasing

Operating systems can 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. 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. 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.

2.2 Physical Page Allocation Case Study

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 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 what body of code required the most time to execute. 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. Past eight processors, the addition of more processors actually slows down the computation and the system begins to exhibit fold-back. Architectural overhead takes up a portion of the execution time as more cores are added. This is believed to be 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 this free list. There are multiple memory zones each with independent lock sets. The Linux kernel rebalances the free lists in bulk to minimize rebalancing time. Even with all of these optimizations, the top level rebalancing 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 that 16 out of the 1000 cores would need to allocate a page at the same time thus causing traffic similar to this test program.

3 Design of a Factored Operating System

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 to applications and system service servers, and maintains a name cache used internally to determine the destination of messages. Applications and system servers execute on top of the microkernel and execute on the same core resources as the 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 core 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. Defaultly each server executes solely on a dedicated processor core. Servers communicate only via the messaging interface provided by the microkernel layer.

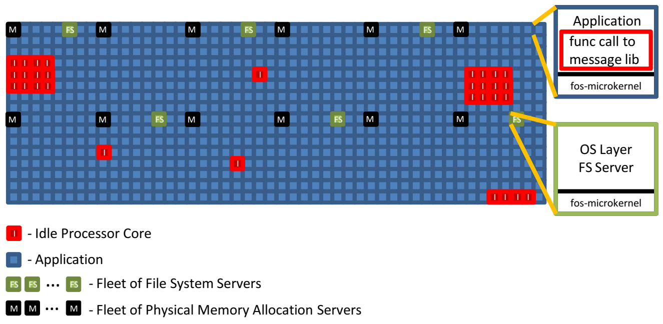

Figure 2. OS and application clients executing on the fos-microkernel

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 can be exposed to the application writer. A small translation library is used to turn system calls into messages from an application layer 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 2 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.

3.1 fos-microkernel

The main functions the fos-microkernel provides are communication between clients executing on cores and resource protection. The fos-microkernel provides a reliable messaging layer to applications and the OS layer. The fos-microkernel also implements the mechanisms of resource protection, but not the protection policy, which it leaves up to the protection manager as implemented in the OS layer.

Each processor core in fos executes a portion of the fos-microkernel. The microkernel on each core is largely independent of the microkernel executing on other cores. By utilizing shared-little design, the microkernel is capable of scaling up to large numbers of processing cores. Microkernels on different cores communicate as needed to transport messages from client tasks.

The fos-microkernel does not have any threads of control running inside of it and does not time-slice with the clients executing on a particular core. Rather the fos-microkernel executes only in response to clients requesting services from it or in response to messages destined for a given core.

3.1.1 Communication

The key service provided by the fos-microkernel to microkernel clients is that of a reliable messaging layer. The fos-microkernel provides a reliable message transport layer. Each fos-microkernel client can allocate a large set of receive mailboxes via which it can receive messages. The receive mailbox can be configured in either a polling mode or can be configured to generate interrupts to the receiving client. A mailbox has a fixed size and is named by the receiving client. Clients send messages by naming a particular destination client and receive mailbox. The fos-microkernel then enqueues the sent message in the receiving client's receive queue. If the receive queue is full, the sending client's send command will return that the receive queue is full and the sender can elect to retry. Message order is guaranteed between any two clients, but is not guaranteed globally.

In addition to transport of messages, the fos-microkernel also maintains a cache of name mapping. The fos-microkernel delegates destination look-up to the name server fleet (running in the OS layer) which maintains the canonical name directory. The name cache provides a manner for microkernel clients to send messages to uniquely

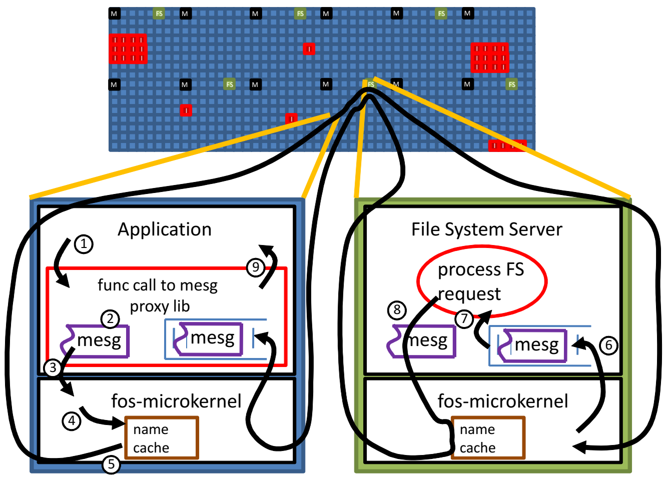

Figure 3. Message walkthrough of an example application file system access.

named clients. The name server also allows for redirection if a client moves to a different processor core. The name server also provides a one-to-many mapping function. This allows for clients to send to a named service which may be implemented by a fleet of servers. The name server can then choose the mapping to a particular server instance based off of physical proximity or load balancing. The name server manager also helps with fault resilience as broken servers can be steered away from when a fault is detected.

Figure 3 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: The message proxy library unpackages the response message and returns data to the application.

3.1.2 Protection and Privilege Isolation

The fos-microkernel does not implement the policies of resource protection, but rather policy decisions are left up to the OS layer. fos differentiates between protection mechanism and policy. The fos-microkernel executes protected operations on behalf of clients, thus it controls the mechanisms of protection, but the policy decisions are delegated to the protection manager. The protection manager is a portion of the OS layer and is implemented in a distributed manner. The fos-microkernel likewise enforces privilege checking to determine if a given client has enough privilege to access a particular resource. The fos-microkernel caches privilege information in a read only manner and calls into the distributed protection manager which is a service provided by the OS layer and is the canonical repository of privilege information. The fos-microkernel restricts which clients can send messages to which other clients. It also restricts access to administrative hardware such as memory protection hardware and I/O messaging hardware. Interface to the memory protection hardware is via a system call interface into the fos-microkernel.

3.1.3 Delegation

The fos-microkernel is designed to delegate functionality to the OS layer in several situations. While this may seem cause cyclic dependencies, the fos-microkernel and delegated clients have been designed with this in mind. Delegation occurs by the microkernel originating messages for client tasks. The client tasks respond via fos-microkernel messaging. In order for dependency loops to not exist, client tasks which can be delegated to by the fos-microkernel are designed to not be reliant on fos-microkernel services which would case a cycle to occur.

An example of microkernel delegation is that of the privilege manager. The privilege manager 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 clients static privileges, so that a privilege fixed point can be reached.

3.1.4 Platform Dependence

One of the goals of fos is to be platform independent. While the authors believe that platform independence is important in the bulk of the code, we do not believe that platform independence extends to the fos-microkernel. The microkernel needs to implement protection mechanisms and as such must be specialized to the hardware it is executing on. I/O interfacing which is handled by the fos-microkernel is also inherently platform dependent. The fos-microkernel must provide high performance messaging to client tasks in order for the OS layer to execute efficiently. Therefore the fos-microkernel design is free to use any hardware provided by the platform to increase the performance of messaging. One example of this is that messaging on shared memory machines utilizes shared memory and memory mapping modification to communicate between cores, while fos's microkernel can use messaging networks on machines which provide hardware messaging networks.

3.2 OS layer

The OS layer provides all of the system services provided by typical modern day operating systems. In contrast to typical modern day operating systems, the OS layer is constructed out of fleets of decentralized servers. In order to provide scalability up to thousands of processor cores, fos's OS layer has been factored first by service being provided. For instance the code which provides file system functionality is not part of the same server as the code which provides physical page allocation. In fact these portions of the operating system run on different spatially disparate cores. Each service is further factored into a fleet of servers. The servers are spatially scattered across the compute fabric. When a application requires a particular service, it contacts the service specific server which it is closest to. Servers execute on disparate computing cores than the applications are executing on and are contacted via fos-microkernel messaging. By not executing portions of the OS on the application processing node, the core that the application is executing on does not have its cache polluted. Also, the application does not need to time-multiplex the compute resource.

Fleets of servers communicate with each other when the OS needs services from other portions of the OS. Fleets of servers communicate via fos-microkernel messaging. An example of two fleets communicating would be when the file system needs physical memory buffers. In this example, the a file system server contacts the physical memory server fleet to request a memory buffer.

Inside of a fleet of servers, the individual servers communicate. fos is structured to reduce this communication, but for shared resources, communication is needed. Servers communicate with each other via fos-microkernel messaging. Servers often need to communicate with multiple of the other servers in the same fleet to complete a transaction. Servers are designed with spatial locality in mind and choose to communicate with the nearest server in the same fleet if possible.

3.2.1 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 thus transaction-oriented. As much as is possible, servers are designed to be stateless. 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

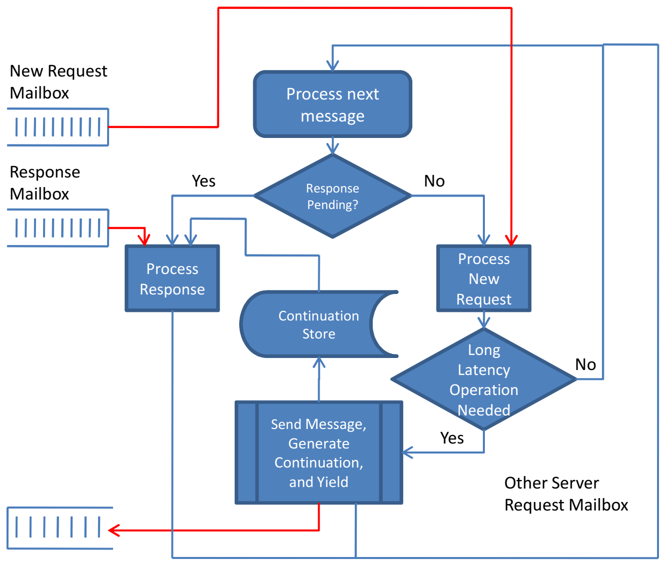

Figure 4. The main runloop for a server.

multiple transactions in sequence. By structuring servers as transaction-oriented stateless processes, server design is much simplified. Also, scalability and robustness is improved as transactions can be routed by the name server manager to differing servers in the same server fleet. Some single stream performance may be sacrificed for this level of robustness and scalability.

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, servers construct up 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 typically structured to process two inbound mailboxes. One for new requests and one for responses. When a ongoing transaction is completed, the server checks the response mailbox. If there are pending responses, the server restarts the needed continuation with the response. Figure 4 shows they typical control flow of a server. Servers are designed to acquire all resources 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 Core Service Servers

The fos OS-layer is composed of many different distributed core services. Following is a list of the servers in the base fos:

- name server
- scheduler/placement server
- physical memory allocation server
- privilege server
- file system server
- driver servers
- networking server

3.4 Naming API

In fos, messaging mailboxes can be textural named. The naming layer is managed by the name servers which are components of the fos OS-layer. On mailbox creation, a name for the mailbox is automatically added to the naming layer. The automatically created name is automatically chosen such that it aids the system in routing messages, therefore more symbolic naming is needed. In order to register a mailbox with a name, an fos client messages the nameserver with a `register_name` message. A `register_name` message includes the mailbox name, which is returned from the creation of a mailbox, the desired name, and flags which indicate whether the named port can be added to a set of mailboxes providing a service. The flags also indicate whether the mailbox is interchangeable with other mailboxes providing the service or whether it is stateful. `register_name` returns whether the name assignment was a success or a failure with a returned error code. Example errors include that the name is already taken or if there are insufficient permissions to create the name.

In addition to `register_name` message, there is also an ability to claim namespaces, via the `claim_namespace` message. By claiming namespaces, the system can guarantee that a namespace will be available for future servers such as system level names. An example of this is that the namespace `"sys:*"` would be reserved by the system on boot which would reserve the entire ”sys” namespace for its use.

Names can be deleted via the `delete_name` message. In order to delete a name where a name points to a set of names, the `delete_specific_name` message is used which allows for removal of specific mappings. When a process is migrated, a specific name would be deleted and a new mapping would be added.

4 Related Work

The work in is motivated by the advent of multicore and manycore processors. If trends continue, we will soon see single chips with 1000's of processor cores. There have been several research projects which have designed prototypes of massively multicore processors. The MIT Raw Processor [25, 22], the Piranha Chip Multiprocessor [6], and the 80-core Intel designed Polaris project [23] are examples of single-chip research multiprocessors. Chip multiprocessor research has begun to transition from research into commercial realization. One example is the Niagara processor [15] designed by Afara Websystems later purchased by Sun Microsystems. The Niagara 1 processor has 8 cores each with four threads. Another commercial massively multicore processor is the TILE64 processor [26] designed by Tilera Corporation. The TILE architecture utilizes a mesh topology for connecting 64 processor cores. The TILE Architecture provides register-mapped on-chip networks to allow cores to explicitly communicate via message passing in addition to using shared memory communication abstraction. The TILE Architecture supports multiple hardware levels of protection and the ability to construct hardwalls which can block communications on the on-chip networks. This research supposes that future processors will look like TILE Architecture processors scaled up to 1000's of cores. In order to facilitate development and to ease porting to x86 compatible processors, this work supposes that future processors will look like the TILE Architecture, but with an industry standard x86_64 instruction set.

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 [16] kernel is another example of a microkernel which attempts to optimize away some of the inefficiencies found in Mach and focuses heavily on performance.

Microkernels have been used in commercial systems. Most notably Mach has been used as the basis of NeXTStep and Mac OS X. The QNX [13] operating system is a commercial microkernel largely used for embedded systems. Also BeOS and the Windows NT kernel are microkernels.

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 function 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. This work embraces the spatial nature of future massively multicore processors and has a scheduler which is not only temporally aware, but also spatially aware. Last, fos, is differentiated from previous microkernels on parallel systems, because the communication costs and sheer number of cores on massively multicore processor is different than in previous parallel systems thus the optimizations made and trade-offs are quite different.

The Tornado [10] operating system which has been extended into the K42 [4] 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 cross-machine shared memory. Also, the scalability claims [5] 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 [8] 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 [7] and Cellular Disco [12]. 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 transiting 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 [24] 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.

This work 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 [21, 20], Sprite [18], and Clouds [9]. 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. This work leverages many of the techniques from distributed Internet scale servers, but instead of applying them to Internet applications, this work applies them on-chip to increase scalability of operating system services.

fos's inspiration for techniques to increase OS scalability is partially derived from different classes of 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 Torrent, Chord, and Freenet. The file system on fos will be inspired by distributed file systems such as AFS [19], OceanStore [14] and the Google File System [11].

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 1,000 - 10,000 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 the factored operating system (fos) which targets 1000+ core multicore systems and replaces traditional time 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, and the NSF.

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] 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.

[4] 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.

[5] 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.

[6] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, pages 282–293, June 2000.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, Apr. 1992.

[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] A. Leon, J. L. Shin, K. Tam, W. Bryg, F. Schumacher, P. Kongetira, D. Weisner, and A. Strong. A power-efficient high-throughput 32-thread SPARC processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, Feb. 2006.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.