Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2010-003 February 8, 2010

An Operating System for Multicore and Clouds: Mechanisms and Implementation

David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

An Operating System for Multicore and Clouds: Mechanisms and Implementation

David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal

{wentzlaf,cg3,beckmann,kmod,abelay,lyouseff,jasonm,agarwal}@csail.mit.edu

CSAIL, Massachusetts Institute of Technology

ABSTRACT

Cloud computers and multicore processors are two emerging classes of computational hardware that have the potential to provide unprecedented compute capacity to the average user. In order for the user to effectively harness all of this computational power, operating systems (OSes) for these new hardware platforms are needed. Existing multicore operating systems do not scale to large numbers of cores, and do not support clouds. Consequently, current day cloud systems push much complexity onto the user, requiring the user to manage individual Virtual Machines (VMs) and deal with many system-level concerns. In this work we describe the mechanisms and implementation of a factored operating system named fos. fos is a single system image operating system across both multicore and Infrastructure as a Service (IaaS) cloud systems. fos tackles OS scalability challenges by factoring the OS into its component system services. Each system service is further factored into a collection of Internet-inspired servers which communicate via messaging. Although designed in a manner similar to distributed Internet services, OS services instead provide traditional kernel services such as file systems, scheduling, memory management, and access to hardware. fos also implements new classes of OS services like fault tolerance and demand elasticity. In this work, we describe our working fos implementation, and provide early performance measurements of fos for both intra-machine and inter-machine operations.

1. INTRODUCTION

The average computer user has an ever-increasing amount of computational power at their fingertips. Users have progressed from using mainframes to minicomputers to personal computers to laptops, and most recently, to multicore and cloud computers. In the past, new operating systems have been written for each new class of computer hardware to facilitate resource allocation, manage devices, and take advantage of the hardware's increased computational capacity. The newest classes of computational hardware, multicore and cloud computers, need new operating systems to take advantage of the increased computational capacity and to simplify user's access to elastic hardware resources.

Cloud computing and Infrastructure as a Service (IaaS) promises a vision of boundless computation which can be tailored to exactly meet a user's need, even as that need grows or shrinks rapidly. Thus, through IaaS systems, users should be able to purchase just the right amount of computing, memory, I/O, and storage to meet their needs at any given time. Unfortunately, counter to the vision, current IaaS systems do not provide the user the same experience as if they were accessing an infinitely scalable multiprocessor computer where resources such as memory, disk, cores, and I/O can all be easily hot-swapped. Instead, current IaaS systems lack system-wide operating systems, requiring users to explicitly manage resources and machine boundaries. If cloud computing is to deliver on its promise, the ease of using a cloud computer must match that of a current-day multiprocessor system.

The next decade will also bring single chip microprocessors containing hundreds or even thousands of computing cores. Making operating systems scale, designing scalable internal OS data structures, and managing these growing resources will be a tremendous challenge. Contemporary OSes designed to run on a small number of reliable cores are not equipped to scale up to thousands of cores or tolerate frequent errors. The challenges of designing an operating system for future multicore and manycore processors are shared with those for designing OSes for current-day cloud computers. The common challenges include scalability, managing elasticity of demand, managing faults, and the challenge of large system programming.

Our solution is to provide a single system image OS, making IaaS systems as easy to use as multiprocessor systems and allowing the above challenges to be addressed in the OS. In this work, we present a factored operating system (fos) which provides a single system image OS on multicore processors as well as cloud computers. fos does so in two steps. First, fos factors system services of a full-featured OS by service. Second, fos further factors and parallelizes each system service into an internet-style collection, or *fleet*, of cooperating servers that are distributed among the underlying cores and machines. All of the system services within fos, and also the fleet of servers implementing each service, communicate via message passing, which maps transparently across multicore computer chips and across cloud computers via networking. For efficiency, when fos runs on shared-memory multicores, the messaging abstraction is implemented using shared memory. Although fos uses the messaging abstraction internally, it does not require applications that it hosts to use message passing for communication. Previous work [26] has introduced the multicore aspects of fos, while this work focuses on how to build an operating system which can service both cloud and multicore computers.

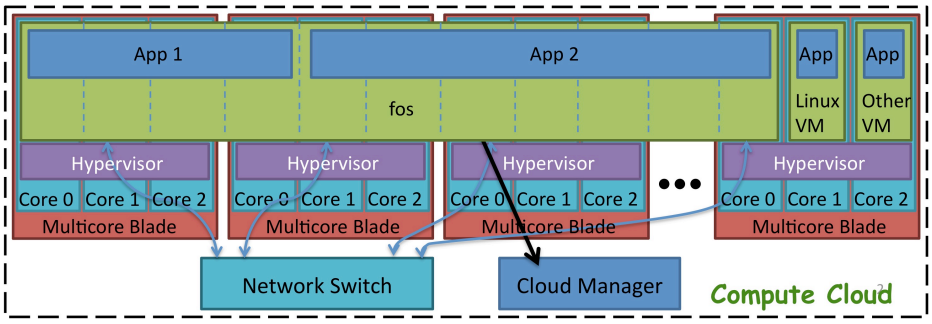

Figure 1: fos provides a single system image across all the cloud nodes.

1.1 Challenges with Current Cloud Systems

Current IaaS systems present a fractured and non-uniform view of resources to the programmer. IaaS systems such as Amazon's EC2 [1] provision resources in units of virtual machines (VM). Using virtual machines as a provisioning unit reduces the complexity for the cloud manager, but without a suitable abstraction layer, this division introduces complexity for the system user. The user of a IaaS system has to worry not only about constructing their application, but also about system concerns such as configuring and managing communicating operating systems. Addressing the system issues requires a completely new skill set than those for application development.

For example, in order for a user to construct an application that can take advantage of more than a single VM, the user application needs to recognize its needs, communicate its needs to the cloud manager, and manage the fractured communication paradigms of intra- and inter-machine communication. For communication inside of a multicore VM, shared memory and pipes are used, while sockets must be used between VMs. The fractured nature of the current IaaS model extends beyond communication mechanisms to scheduling and load balancing, system administration, I/O devices, and fault tolerance. For system administration, the user of an IaaS cloud system needs to manage a set of different computers. Examples of the system administration headaches include managing user accounts within a machine versus externally via NIS or Kerberos, managing processes between the machines (using 'ps' and 'kill' within a machine, and a batch queue or ad-hoc mechanisms between machines), and keeping configuration files and updates synchronized between machines (cfengine) versus within one machine. There is no generalized way to access remote I/O devices on operating systems such as Linux. Point solutions exist for differing I/Os, for instance NFS for disk, and VNC for display. Last, faults are accentuated in a VM environment because the user has to manage cases where a whole VM crashes as a separate case from a process which has crashed.

Scheduling and load balancing differs substantially within and between machines as well. Existing operating systems handle scheduling within a machine, but the user must often build or buy server load balancers for scheduling across machines. Cloud aggregators and middleware such as RightScale [22] and Amazon's CloudWatch Auto Scaling [1] provide automatic cloud management and load balancing tools, but they are typically application-specific and tuned to web application serving.

1.2 Benefits of a Single System Image

fos proposes to provide a single system image across multicores and the cloud as shown in Figure 1. This abstraction can be built on top of VMs which are provided by an IaaS service or directly on top of a cluster of machines. A single system image has the following advantages over the ad-hoc approach of managing VMs each running distinct operating system instances:

- Ease of administration: Administration of a single OS is easier than many machines. Specifically, OS update, configuration, and user management are simpler.
- Transparent sharing: Devices can be transparently shared across the cloud. Similarly, memory and disk on one physical machine can transparently be used on another physical machine (*e.g.*, paging across the cloud)
- Informed optimizations: OS has local, low-level knowledge, thereby allowing it to make better, finer-grained optimizations than middleware systems.
- Consistency: OS has a consistent, global view of process management and resource allocation. Intrinsic load balancing across the system is possible, and so is easy process migration between machines based on load, which is challenging with middleware systems. A consistent view also enables seamless scaling, since application throughput can be scaled up as easily as exec'ing new processes. Similarly, applications have a consistent communication and programming model whether the application resides inside of one machine or spans multiple physical machines. Furthermore, debugging tools are uniform across the system, which facilitates debugging multi-VM applications.
- Fault tolerance: Due to global knowledge, the OS can take corrective actions on faults.

This paper describes a working implementation of a prototype factored operating system, and presents early performance measurements on fos operations within a machine and between machines. Our fos prototype provides a single system image across multicores and clouds, and includes a microkernel, messaging layer, naming layer, protected memory management, a local and remote process spawning interface, a file system server, a block device driver server, a message proxy network server, a basic shell, a webserver, and a network stack.

This paper is organized as follows. Section 2 details the common challenges that cloud and multicore operating systems face. Section 3 explores the architecture of fos. Section 4 explores the detailed implementation of fos on clouds and multicores through some example OS operations. Section 5 measures the current fos

prototype implementation on multicore and cloud systems. Section 6 places fos in context with previous systems. And finally we conclude.

2. MULTICORE AND CLOUD OPERATING SYSTEM CHALLENGES

Cloud computing infrastructure and manycore processors present many common challenges with respect to the operating system. This section introduces what we believe are the main problems OS designers will need to address in the next decade. Our solution, fos, seeks to address these challenges in a solution that is suitable for both multicore and cloud computing.

2.1 Scalability

The number of transistors which fit onto a single chip microprocessor is exponentially increasing [18]. In the past, new hardware generations brought higher clock frequency, larger caches, and more single stream speculation. Single stream performance of microprocessors has fallen off the exponential trend [5]. In order to turn increasing transistor resources into exponentially increasing performance, microprocessor manufacturers have turned to integrating multiple processors onto a single die [27, 25]. Current OSes were designed for single processor or small number of processor systems. The current 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. Therefore multicore OSes need to embrace scalability and make it a first order design constraint. In our previous work [26], we investigated the scalability limitations of contemporary OS design including: locks, locality aliasing, and reliance on shared memory.

Concurrent with the multicore revolution, cloud computing and IaaS systems have been gaining popularity. This emerging computing paradigm has a huge potential to transform the computing industry and programming models [7]. The number of computers being added by cloud computing providers has been growing at a vast rate, driven largely by user demand for hosted computing platforms. The resources available to a given cloud user is much higher than is available to the non-cloud user. Cloud resources are virtually unlimited for a given user, only restricted by monetary constraints. Example public clouds and IaaS services include Amazon's EC2 [1] and Rackspace's Cloud Server [2]. Thus, it is clear that scalability is a major concern for future OSes in both single machine and cloud systems.

2.2 Variability of Demand

We define elasticity of resources as the aspect of a system where the available resources can be changed dynamically over time. By definition, manycore systems provide a large number of general-purpose, schedulable cores. Furthermore, the load on a manycore system translates into number of cores being used. Thus the system must manage the number of live cores to match the demand of the user. For example, in a 1,000-core system, the demand can require from 1 to 1,000 cores. Therefore, multicore OSes need to manage the number of live cores which is in contrast to single core OSes which only have to manage whether a single core is active or idle.

In cloud systems, user demand can grow much larger than in the past. Additionally, this demand is often not known ahead of time by the cloud user. It is often the case that users wish to handle peak load without over-provisioning. In contrast to cluster systems where the number of cores is fixed, cloud computing makes more resources available on-demand than was ever conceivable in the past.

A major commonality between cloud computing and multicore systems is that the demand is not static. Furthermore, the variability of demand is much higher than in previous systems and the amount of available resources can be varied over a much broader range in contrast to single-core or fixed-sized cluster systems.

The desire to reach optimal power utilization forces current system designers to match the available resources to the demand. Heat and energy consumption impact computing infrastructure from chip design all the way up to the cost of running a data center. As a result, fos seeks to reduce the head production and power consumption while maintaining the throughput requirements imposed by the user.

2.3 Faults

Managing software and hardware faults is another common challenge for future multicore and cloud systems. In multicore systems, hardware faults are becoming more common. As the hardware industry is continuously decreasing the size of transistors and increasing their count on a single chip, the chance of faults is rising. With hundreds or thousands of cores per chip, system software components must gracefully support dying cores and bit flips. In this regard, fault tolerance in modern OSes designed for multicore is becoming an essential requirement.

In addition, faults in large-scale cloud systems are common. Cloud applications usually share cloud resources with other users and applications in the cloud. Although each users' application is encapsulated in a virtual container (for example, a virtual machine in an EC2 model), performance interference from other cloud users and applications can potentially impact the quality of service provided to the application.

Programming for massive systems is likely to introduce software faults. Due to the inherent difficulty of writing multithreaded and multiprocess applications, the likelihood of software faults in those applications is high. Furthermore, the lack of tools to debug and analyze large software systems makes software faults hard to understand and challenging to fix. In this respect, dealing with software faults is another common challenge that OS programming for multicore and cloud systems share.

2.4 Programming Challenges

Contemporary OSes which execute on multiprocessor systems have evolved from uniprocessor OSes. This evolution was achieved by adding locks to the OS data structures. There are many problems with locks, such as choosing correct lock granularity for performance, reasoning about correctness, and deadlock prevention. Ultimately, programming efficient large-scale lock-based OS code is difficult and error prone. Difficulties of using locks in OSes is discussed in more detail in [26].

Developing cloud applications composed of several components deployed across many machines is a difficult task. The prime reason for this is that current IaaS cloud systems impose an extra layer of indirection through the use of virtual machines. Whereas on multiprocessor systems the OS manages resources and scheduling, on cloud systems much of this complexity is pushed into the application by fragmenting the application's view of the resource pool.

Furthermore, there is not a uniform programming model for communicating within a single multicore machine and between machines. The current programming model requires a cloud programmer to write a threaded application to use intra-machine resources while socket programming is used to communicate with components of the application executing on different machines.

In addition to the difficulty of programming these large-scale hierarchical systems, managing and load-balancing these systems is

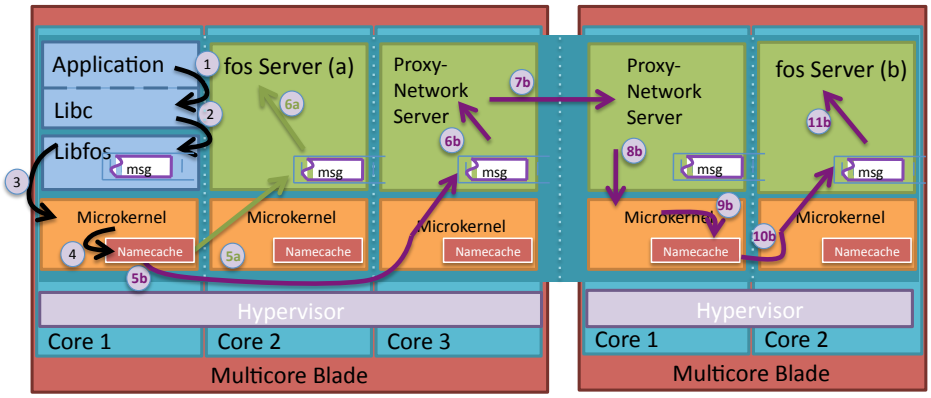

Figure 2: An overview of the fos servers architecture, highlighting the cross-machine interaction between servers in a manner transparent to the application. In scenario (a), the application is requesting services from fos server a which happens to be local to the application. In scenario (b), the application is requesting service which is located on another machine.

proving to be a daunting task as well. Ad-hoc solutions such as hardware load-balancers have been employed in the past to solve such issues. These solutions are often limited to a single level of the hierarchy (at the VM level). In the context of fos, however, this load balancing can be done inside the system, in a generic manner (i.e. one that works on all messaging instead of only tcp/ip traffic) and on a finer granularity than at the VM or single machine level. Furthermore, with our design, the application developer need not be aware of such load balancing.

Scalability, elasticity of demand, faults, and difficulty in programming large systems are common issues for emerging multicore and cloud systems.

3. ARCHITECTURE

fos is an operating system which takes scalability and adaptability as *the* first order design constraints. Unlike most previous OSes where a subsystem scales up to a given point, beyond which the subsystem must be redesigned, fos ventures to develop techniques and paradigms for OS services which scale from a few to thousands of cores. In order to achieve the goal of scaling over multiple orders of magnitude in core count, fos uses the following design principles:

- *Space multiplexing replaces time multiplexing.* Due to the growing bounty of cores, there will soon be a time where the number of cores in the system exceeds the number of active processes. At this point scheduling becomes a layout program, not a time-multiplexing problem. The operating system will run on distinct cores from the application. This gives spatially partitioned working sets; the OS does not interfere with the application's cache.
- *OS is factored into function-specific services, where each is implemented as a parallel, distributed service.* In fos, services collaborate and communicate only via messages, although applications can use shared memory if it is supported. Services are bound to a core, improving cache locality. Through a library layer, libfos, applications communicate to services via messages. Services themselves leverage ideas from collaborating Internet servers.
- *OS adapts resource utilization to changing system needs.* The utilization of active services is measured, and highly loaded services are provisioned more cores (or other resources). The OS closely manages how resources are used.
- *Faults are detected and handled by OS.* OS services are monitored by watchdog process. If a service fails, a new instance is spawned to meet demand, and the naming service reassigns communication channels.

The following sections highlight key aspects of the fos architecture, shown in Figure 2. fos runs across multiple physical machines in the cloud. In the figure, fos runs on an IaaS system on top of a hypervisor. A small microkernel runs on every core, providing messaging between applications and servers. The global name mapping is maintained by a distributed set of proxy-network servers that also handle inter-machine messaging. A small portion of this global namespace is cached on-demand by each microkernel. Applications communicate with services through a library layer (libfos), which abstracts messaging and interfaces with system services.

3.1 Microkernel

fos is a microkernel operating system. The fos microkernel executes on every core in the system. fos uses a minimal microkernel OS design where the microkernel only provides a protected messaging layer, a name cache to accelerate message delivery, basic time multiplexing of cores, and an Application Programming Interface (API) to allow the modification of address spaces and thread creation. All other OS functionality and applications execute in user space. OS system services execute as userland processes, but may possess capabilities to communicate with other system services which user processes do not.

Capabilities are extensively used to restrict access into the protected microkernel. The memory modification API is designed to allow a process on one core to modify the memory and address space on another core if appropriate capabilities are held. This approach allows fos to move significant memory management and scheduling logic into userland space.

3.2 Messaging

One operating system construct that is necessary for any multicore or cloud operating system is a form of inter-process communication and synchronization. fos solves this need by providing a simple process-to-process messaging API. There are several key advantages to using messaging for this mechanism. One advantage is the fact that messaging can be implemented on top of shared

memory, or provided by hardware, thus allowing this mechanism to be used for a variety of architectures. Another advantage is that the sharing of data becomes much more explicit in the programming model, thus allowing the programmer to think more carefully about the amount of shared data between communicating processes. By reducing this communication, we achieve better encapsulation as well as scalability, both desirable traits for a scalable cloud or multicore operating system.

Using messaging is also beneficial in that the abstraction works across several different layers without concern from the application developer. To be more concrete, when one process wishes to communicate with another process it uses the same mechanism for this communication regardless of whether the second process is on the same machine or not. Existing solutions typically use a hierarchical organization where intra-machine communication uses one mechanism while inter-machine communication uses another, often forcing the application developer to choose a-priori how they will organize their application around this hierarchy. By abstracting this communication mechanism, fos applications can simply focus on the application and communication patterns on a flat communication medium, allowing the operating system to decide whether or not the two processes should live on the same VM or not. Additionally, existing software systems which rely on shared memory are also relying on the consistency model and performance provided by the underlying hardware.

fos messaging works intra-machine and across the cloud, but uses differing transport mechanisms to provide the same interface. On a shared memory multicore processor, fos uses message passing over shared memory. When messages are sent across the cloud, messages are sent via shared memory to the local proxy server which then uses the network (*e.g.*, Ethernet) to communicate with a remote proxy server which then delivers the message via shared memory on the remote node.

Each process has a number of mailboxes that other processes may deliver messages to provided they have the credentials. fos presents an API that allows the application to manipulate these mailboxes and their properties. An application starts by creating a mailbox. Once the mailbox has been created, capabilities are created which consist of keys that may be given to other servers allowing them to write to the mailbox.

In addition to mailbox creation and access control, processes within fos are also able to register a mailbox under a given name. Other processes can then communicate with this process by sending a message to that name and providing the proper capability. The fos microkernel and proxy server assume the responsibility of routing and delivering said message regardless of whether or not the message crosses machine boundaries.

3.3 Naming

One unique approach to the organization of multiple communicating processes that fos takes is the use of a naming and lookup scheme. As mentioned briefly in the section on messaging, processes are able to register a particular name for their mailbox. This namespace is a hierarchical URI much like a web address or filename. This abstraction provides great flexibility in load balancing and locality to the operating system.

The basic organization for many of fos's servers is to divide the service into several independent processes (running on different cores) all capable of handling the given request. As a result, when an application messages a particular service, the nameserver will provide a member of the fleet that is best suited for handling the request. To accomplish this, all of the servers within the fleet register under a given name. When a message is sent, the nameserver will provide the server that is optimal based on the load of all of the servers as well as the latency between the requesting process and each server within the fleet.

While much of the naming system is in a preliminary stage, we have various avenues to explore for the naming system. When multiple servers want to provide the same service, they can share a name. We are investigating policies for determining the correct server to route the message to. One solution is to have a few fixed policies such as round robin or closest server. Alternatively, custom policies could be set via a callback mechanism or complex load balancer. Meta-data such as message queue lengths can be used to determine the best server to send a message to.

As much of the system relies on this naming mechanism, the question of how to optimally build the nameserver and manage caching associated with it is also a challenging research area that will be explored. This service must be extremely low latency while still maintaining a consistent and global view of the namespace. In addition to servers joining and leaving fleets on the fly, thus requiring continual updates to the namelookup, servers will also be migrating between machines, requiring the nameserver (and thus routing information) to be updated on the fly as well. The advantage to this design is that much of the complexity dealing with separate forms of inter-process communication in traditional cloud solutions is abstracted behind the naming and messaging API. Each process simply needs to know the name of the other processes it wishes to communicate with, fos assumes the responsibility of efficiently delivering the message to the best suited server within the fleet providing the given service. While a preliminary flooding based implementation of the namserver is currently being used, the long term solution will incorporate ideas from P2P networking like distributed hash tables as in Chord [23] and Coral [13].

3.4 OS Services

A primary challenge in both cloud computing and multicore is the unprecedented scale of demand on resources, as well as the extreme variability in the demand. System services must be both scalable and *elastic*, or dynamically adaptable to changing demand. This requires resources to shift between different system services as load changes.

fos addresses these challenges by parallelizing each system service into a *fleet* of spatially-distributed, cooperating servers. Each service is implemented as a set of processes that, in aggregate, provide a particular service. Fleet members can execute on separate machines as well as separate cores within a machine. This improves scalability as more processes are available for a given service and improves performance by exploiting locality. Fleets communicate internally via messages to coordinate state and balance load. There are multiple fleets active in the system: *e.g.*, a file system fleet, a naming fleet, a scheduling fleet, a paging fleet, a process management fleet, et cetera.

Assuming a scalable implementation, the fleet model is elastic as well. When demand for a service outstrips its capabilities, new members of the fleet are added to meet demand. This is done by starting a new process and having it handshake with existing members of the fleet. In some cases, clients assigned to a particular server may be reassigned when a new server joins a fleet. This can reduce communication overheads or lower demand on local resources (*e.g.*, disk or memory bandwidth). Similarly, when demand is low, processes can be eliminated from the fleet and resources returned to the system. This can be triggered by the fleet itself or an external watchdog service that manages the size of the fleet. A key research question is what are the best policies for growing, shrinking, and layout (scheduling) of fleets.

Fleets are an elegant solution to scalability and elasticity, but are complicated to program compared to straight-line code. Furthermore, each service may employ different parallelization strategies and have different constraints. fos addresses this by providing (i) a cooperative multithreaded programming model; (ii) easy-to-use remote procedure call (RPC) and serialization facilities; and (iii) data structures for common patterns of data sharing.

3.4.1 fos Server Model

fos provides a server model with cooperative multithreading and RPC semantics. The goal of the model is to abstract calls to independent, parallel servers to make them appear as local libraries, and to mitigate the complexities of parallel programming. The model provides two important conveniences: the server programmer can write simple straight-line code to handle messages, and the interface to the server is simple function calls.

Servers are event-driven programs, where the events are messages. Messages arrive on one of three inbound mailboxes: the external (public) mailbox, the internal (fleet) mailbox, and the response mailbox for pending requests. To avoid deadlock, messages are serviced in reverse priority of the above list.

New requests arrive on the external mailbox. The thread that receives the message is now associated with the request and will not execute any other code. The request may require communication with other servers (fleet members or other services) to be completed. Meanwhile, the server must continue to service pending requests or new requests. The request is processed until completion or a RPC to another service occurs. In the former case, the thread terminates. In the latter, the thread yields to the cooperative scheduler, which spawns a new thread to wait for new messages to arrive.

Requests internal to the fleet arrive on the internal mailbox. These deal with maintaining data consistency within the fleet, load balancing, or growing/shrinking of the fleet as discussed above. Otherwise, they are handled identically to requests on the external mailbox. They are kept separate to prevent others from spoofing internal messages and compromising the internal state of the server.

Requests on the response mailbox deal with pending requests. Upon the receipt of such a message, the thread that initiated the associated request is resumed.

The interface to the server is a simple function call. The desired interface is specified by the programmer in a header file, and code is generated to serialize these parameters into a message to the server. Likewise, on the receiving end, code is generated to deserialize the parameters and pass them to the implementation of the routine that runs in the server. On the "caller" side, the thread that initiates the call yields to the cooperate scheduler. When a response arrives from the server, the cooperative scheduler will resume the thread.

This model allows the programmer to write straight-line code to handle external requests. There is no need to generate complex state machines or split code upon interaction with other servers, as the threading library abstracts away the messaging. However, this model doesn't eliminate all the complexities of parallel programming. Because other code will execute on the server during an RPC, locking is still at times required, and the threading library provides mechanisms for this.

The cooperative scheduler runs whenever a thread yields. If there are threads ready to run (*e.g.*, from locking), then they are scheduled. If no thread is ready, then a new thread is spawned that waits on messages. If threads are sleeping for too long, then they are resumed with a time out error status.

The model is implemented as a user-space threading library written in C and a C-code generation tool written in python. The code generator uses standard C header files with a few custom preprocessor macros. Unlike some similar systems, there is no custom language. The server writer is responsible only for correctly implementing the body of each declared function and providing any special serialization/deserialization routines, if necessary.

3.4.2 Parallel Data Structures

One key aspect to parallelizing operating system services is managing state associated with a particular service amongst the members of the fleet. As each service is quite different in terms of resource and performance needs as well as the nature of requests, several approaches are required. While any given fleet may choose to share state internally using custom mechanisms, a few general approaches will be provided for the common case.

One solution is to employ a restful design, borrowing ideas from many Internet services [12]. In this organization, each of the servers are stateless and all of the information needed to perform a particular service is passed in to the server with the given request. This approach is advantageous in that each of the servers is independent and many can easily be spawned or destroyed on the fly with little to no interaction between servers required for managing state. The drawback is that all of the state is stored at the client, which can limit some of the control that the server has over that data along with allowing the client to alter the data.

Another solution fos plans to employ is a managed data backing store. In this solution, the operating system and support libraries provide an interface for restoring and retrieving data. On the back-end, each particular server stores some of the data (acting as a cache) and communicates with other members of the fleet for the state information not homed locally. There are existing solutions to this problem in the P2P community [23, 13] that we plan to explore that will leverage locality information. Special care needs to be taken to handle joining and removing servers from a fleet. By using a library provided by the operating system and support libraries, the code to manage this distributed state can be tested and optimized, alleviating the application developer from concerning themselves with consistency of distributed data.

There are several solutions for managing shared and distributed state information. The important aspect of this design is that computation is decoupled from the data, allowing the members of a fleet to be replicated on the fly to manage changing load.

4. CASE STUDIES

This section presents detailed examples of key components of fos. It both illustrates how fos works and demonstrates how fos solves key challenges in the cloud.

4.1 File System

An example of the interaction between the different servers in fos is the fos file server. Figure 3 depicts an anatomy of a file system access in fos. In this figure, the application client, the fos file system server and the block device driver server are all executing on distinct cores to diminish the cache and performance interferences among themselves. Since the communication between the application client and systems servers, and amongst the system servers, is via the fos messaging infrastructure, proper authentication and credentials verifications for each operation is performed by the messaging layer in the microkernel. This example assumes all services are on the same machine, however the multi-machine is a logical extension to this example, with a proxy server bridging the messaging abstraction between the two machines.

fos intercepts the POSIX file system calls in order to support compatibility with legacy POSIX applications. It bundles the POSIX

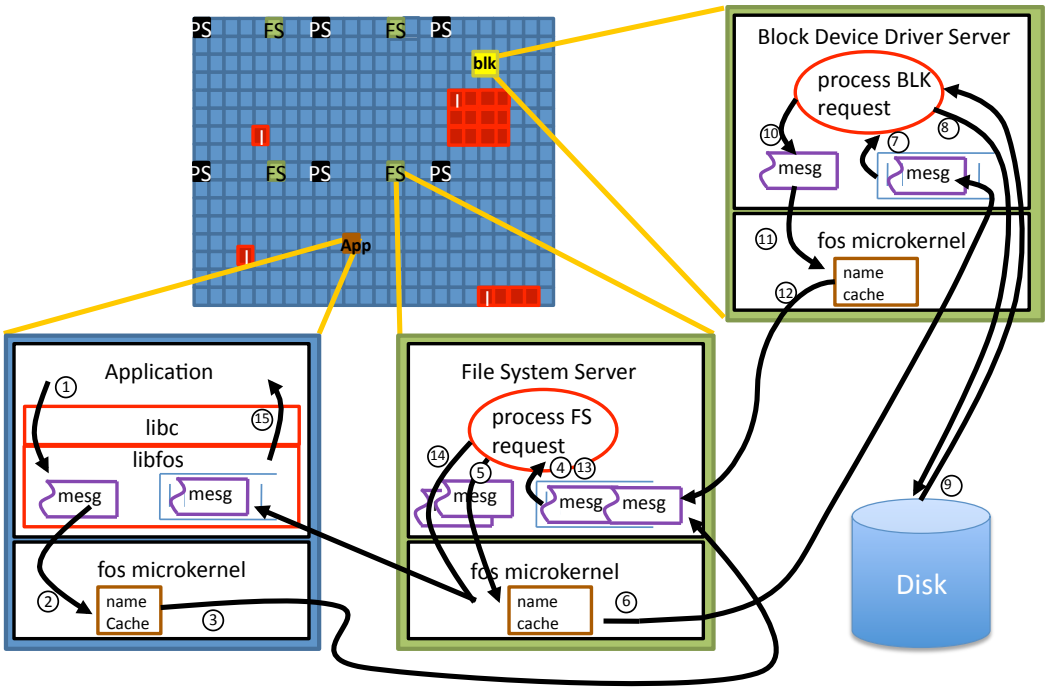

Figure 3: Anatomy of a File System Access

calls into a message and sends it to the file system server. The microkernel determines the destination server of the message and verifies the capabilities of the client application to communicate with the server. It, then look up the destination server in its name cache and finds out which core it is executing on. If the server is a local server (i.e. executing on the same machine as the application), the microkernel forwards the message to the destination application. In Figure 3, fos intercepts the application File system access in step 1, bundles it in a message in step 2 to be sent via the messaging layer. Since the destination server for this message is the file system server, fos queries the name cache and sends the message to the destination core in step 3.

Once the file system server receives a new message in its incoming mailbox queue, it services the request. If the data requested by the application is cached, the server bundles it into a message and sends it back to the requesting application. Otherwise, it fetches the needed sectors from disk through the block device driver server. In the file system anatomy figure, step 5 represents the bundling of the sectors request into *block messages* while step 6 represents the look-up of the block device driver in the name cache. Once the server is located, the fos microkernel places the message in the incoming mailbox queue of the block device driver server as shown in step 6.

The block device driver server provides Disk I/O operations and access to the physical disk. In response to the incoming message, the block device driver server processes the request enclosed in the incoming message, fetches the sectors from disk as portrayed in steps 7, 8 and 9 respectively in the figure. Afterward, it encapsulates the fetched sectors in a message and sends it back to the file system server, as in steps 10, 11 and 12. In turn, the file server processes the acquired sectors from the incoming mailbox queue, encapsulates the required data into messages and send them back to the client application. In the client application, the *libfos* receives the data at its incoming mailbox queue and processes it in order to provide the file system access requested by the client application. These steps are all represented by steps 13 through 15 in the file system access anatomy in figure 3.

Libfos provides several functions, including compatibility with POSIX interfaces. The user application can either send the file system requests directly through the fos messaging layer or through *libfos*. In addition, if the file system server is not running on the local machine (i.e. the name cache could not locate it), the message is forwarded to the proxy server. The proxy server has the name cache and location of all the remote servers. In turn, it determines the appropriate destination machine of the message, bundles it into a network message and sends it via the network stack to the designated machine. Although this adds an extra hop through the proxy server, it provides the system with the transparency for accessing local or remote servers, without requiring any application or server modification. In a cloud environment, the uniform messaging and naming allows servers to be assigned to any machine in the system thereby providing a single system image, instead of the fragmented view of the cloud resources. It also provides a uniform application programming model to use inter-machine and intra-machine resources in the cloud.

4.2 Spawning Servers

To expand a fleet by adding a new server, one must first spawn the new server process. Spawning a new process proceeds much like in a traditional operating system, except in fos, this action needs to take into account the machine on which the process should be spawned. Spawning begins with a call to the spawnProcess() function; this arises through an intercepted 'exec' syscall from our POSIX compatibility layer, or by directly calling the spawnProcess function by a fos-aware application. By directly calling the spawnProcess function, parent processes can exercise greater control over where their children are placed by specifying constraints on what machine to run on, what kinds of resources the child will need, and

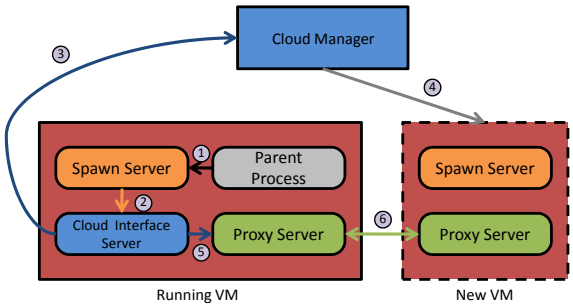

Figure 4: Spawning a VM

locality hints to the scheduler.

The spawnProcess function bundles the spawn arguments into a message, and sends that message to the spawn server's incoming request mailbox. The spawn server must first determine which machine is most suitable for hosting that process. It makes this decision by considering the available load and resources of running machines, as well as the constraints given by the parent process in the spawnProcess call. The spawn server interacts with the scheduler to determine the best machine and core for the new process to start on. If the best machine for the process is the local machine, the spawn server sets up the address space for the new process and starts it. The spawn server then returns the PID to the process that called spawnProcess by responding with a message. If the scheduler determined that spawning remote is best, the spawn server forwards the spawn request to the spawn server on the remote machine, which then spawns the process.

If the local spawn server was unable to locate a suitable machine to spawn the process, it will initiate the procedure of spawning a new VM. To do this, it sends a message to the cloud interface server, describing what resources the new machine should have; when the cloud interface server receives this message, it picks the best type of VM to ask for. The cloud interface server then spawns the new VM by sending a request to the cloud manager via Internet requests (the server outside of fos which is integrated into the underlying cloud infrastructure eg. EC2). When the cloud manager returns the VM ID, the cloud interface server waits until the new VM acquires an IP address. At this point, the cloud interface server begins integration of the new VM into the fos single system image.

The newly-booted VM starts in a bare state, waiting for the spawner VM to contact it. The cloud interface server notifies the local proxy server that there is a new VM at the given IP address that should be integrated into the system, and the proxy server then connects to the remote proxy server at that IP and initiates the proxy bootstrap process. During the bootstrap process, the proxy servers exchange current name mappings, and notify the rest of the machines that there is a new machine joining the system. When the local proxy server finishes this setup, it responds to the cloud interface server that the VM is fully integrated. The cloud interface server can then respond to the local spawn server to inform it that there is a new machine that is available to spawn new jobs, which then tells all the spawn servers in the fleet that there is a new spawn server and a new machine available. The local spawn server finally then forwards the original spawn call to the remote spawn server on the new VM.

In order to smooth the process of creating new VMs, the spawning service uses a pair of high- and low-water-marks, instead of spawning only when necessary. This allows the spawning service to mask VM startup time by peemptively spawning a new VM when the resources are low but not completely depleted. It also prevents the ping-ponging effect, where new VMs are spawned and destroyed unnecessarily when the load is near the new-VM threshold, and gives the spawn servers more time to communicate with each other and decide whether a new VM needs to be spawned.

4.3 Elastic Fleet

As key aspects of the design of fos include scalability and adaptability, this section serves to describe how a fleet grows to match demand. If, while the system is running, the load changes, then the system should respond in a way that meets that demand if at all possible. In the context of a fos fleet, if the load become too high for the fleet to handle requests at the desirable rate, then a watchdog process for the fleet can grow the fleet. The watchdog does this by spawning a new member of the fleet and initiating the handshaking process that allows the new server to join the fleet. During the handshaking process, existing members of the fleet are notified of the new member, and state is shared with the new fleet member. Additionally, the scheduler may choose to spatially re-organize the fleet so as to reduce the latency between fleet members and those processes that the fleet is servicing.

As a concrete example, if there are many servers on a single machine that are all requesting service look-ups from the nameserver, the watchdog process may notice that all of the queues are becoming full on each of the nameservers. It may then decide to spawn a new nameserver and allow the scheduler to determine which core to put this nameserver on so as to alleviate the higher load.

While similar solutions exist in various forms for existing IaaS solutions, the goal of fos is to provide the programming model, libraries and runtime system that can make this operation transparent. By using the programming model provided for OS services as well as the parallel data structures for backing state, many servers can easily enjoy the benefit of being dynamically scalable to match demand.

While the mechanism for growing the fleet will be generic, there are several aspects of this particular procedure that will be service specific. One issue that arises is obtaining the metadata required to make this decision and choosing the policy over that metadata to define the decision boundary. To solve this issue, the actual policy can be provided by members of the fleet.

The fact that this decision is made by part of the operating system is a unique and advantageous difference fos has over existing solutions. In particular, the fleet expansion (and shrinking) can be a global decision based on the health and resources available in a global sense, taking into consideration the existing servers, their load and location (latency) as well as desired throughput or monetary concerns from the system owner. By taking all of this information into consideration when making the scaling scheduling decision, fos can make a much more informed decision than solutions that simply look at the cloud application at the granularity of VMs.

5. RESULTS AND IMPLEMENTATION

fos has been implemented as a Xen para-virtualized machine (PVM) OS. We decided to implement fos as a PVM OS in order to support the cloud computing goals of this project, as this allows us to run fos on the EC2 and Eucalyptus cloud infrastructure [20]. It also simplifies the driver model as the Xen PVM interface abstracts away many of the details of particular hardware. fos and its underlying design does not require a hypervisor but our implementation uses a hypervisor out of convenience and in order to be able

	fos	Linux
min	12169	1321
avg	13327	1328
max	28548	9985
stdev	412.1	122.8

Table 1: local syscall time – intra-machine echo (in cycles)

	fos	Linux
min	4.00	0.199
avg	9.66	0.274
max	85.0	0.395
stddev	15.8	0.0491

Table 2: remote syscall time – inter-machine echo (in ms)

	fos	Linux
min	0.049	0.017
avg	0.065	0.032
max	0.116	0.064
mdev	0.014	0.009

Table 3: ping response (in ms)

to experiment with our system in real cloud IaaS infrastructures.

fos is currently a preemptive multitasking multiprocessor OS executing on real x86_64 hardware. We have a working microkernel, messaging layer, naming layer, protected memory management, a spawning interface, a basic system server, a file system server supporting ext2, a block device driver server, a message proxy server and a full network stack via lwIP[11]. Furthermore, we have a multi-machine cloud interface server which interacts with Eucalyptus to spawn new VMs on our testbed cluster. In addition, we have developed several applications for fos as a proof-of-concept, including a basic shell and a web-server. We are now expanding our collection of system servers and optimizing our entire system performance.

Our hardware testbed cluster is composed of a 16-machine cluster; each machine consists of two Intel Xeon X5460 processors for a total of 8 cores running at 3.16GHz processors. Each machine has 8GB of main memory. Furthermore, the machines are interconnected via two 1 Gbps Ethernet ports which are bonded.

In this section, we present some preliminary results that we have been gathering from our system. However, a key component of our current work is performance optimizations to make fos competitive with Linux in these basic metrics as well as in the cloud. We strongly believe that we will obtain significant performance improvement by the camera-ready deadline of this paper.

5.1 System Calls

This section explores the performance of null system calls, a basic metric of operating system overhead and performance.

5.1.1 Local

fos depends heavily on messaging performance between local cores. In a traditional monolithic kernel, system calls are handled on a local core through trap-and-enter. In contrast, fos distributes its OS services across independent cores and accesses them with message passing. The following benchmarks compares the traditional monolithic system calls of Linux with null message echos in fos.

Preliminary results shown in Table 1 show that fos's unoptimized message passing is slower than the Linux system call implementation. It is expected that this gap will be much narrower as fos development continues.

5.1.2 Remote

Remote system calls consist mainly of a roundtrip communication between two machines. The remote syscall benchmark shown in Table 2 is used to determine the overhead for two processes to communicate with each other when they live on different VMs. This benchmark serves to measure the preliminary speed of the communication pathway between two fos applications residing on separate machines. The data has to pass through the following servers, in order: proxy, network stack, network interface, (over the wire), network interface, network stack, proxy server, echo application and likewise the same servers in reverse before the message makes it back to the original sending application.

It is our goal to optimize and reduce this number as much as possible. However, it is important to note that latency for on-chip communication is expected to be on the order of nano-seconds whereas inter-machine communication will be on the order of milli-seconds. As such, the co-location of processes with data will be crucial for high performance systems in the cloud.

It is also important to note that with this particular test, there is a sequential order through the list of servers that the data must pass through, essentially forming a pipeline. If we allow several outstanding in-flight packets through the system (thus trying to keep the pipeline more full) the throughput can be increased without affecting the latency.

For comparison, we collected results on a similar echo benchmark, using network socket connections between a pair of Linux systems.

5.2 Ping Response

The ping response time is used to determine the overhead of data going through the virtualized network interface, network stack and then back out of the network interface. Essentially this benchmark focuses on the overhead of the network interface server and the network stack without involving any other applications or servers. The results in table 3 were gathered by spawning an instance of fos, then pinging it from the same machine to avoid network latency. The min is within range of existing solutions. The standard deviation and max are a bit high, however we believe further optimizations can reduce both of these metrics. Our current implementation uses a single fos server for the network interface and a single fos server for the network stack. In the future we plan to factor the network stack into a server fleet, where each server within the fleet will be responsible for a subset of the total number of flows in the system.

For reference we have also included the ping times for running Linux in a DomU VM with the same setup.

5.3 Process Creation

fos implements a fleet of spawning servers. This fleet is used to spawn processes both locally (same machine) and remotely (different machine). Table 4 shows the process creation time for each case. Time is measured from sending the request to the spawn server to receipt of the response. The spawn server does *not* wait for the process to complete or be scheduled; it responds immediately after enqueuing the new process at the scheduler. The numbers in Table 4 are collected over twenty-five spawns of a dummy application. The reduced number of runs is why these results show less variance than, say Table 3. A remote spawn involves additional messages within the spawn fleet to forward the request, as well as

	Local	Remote
min	2.0	12.8
avg	2.6	20.0
max	6.7	32.0

Table 4: fos process creation time in ms.

	fos	Linux
min	2.28	0.245
avg	3.07	0.257
max	3.65	0.264
stddev	0.38	0.006

Table 5: Web server request time (ms)

inter-machine overhead. Therefore remote spawns require $10\times$ as much time to occur.

5.4 File System

We currently provide an *ext2* file system implementation in fos. Upon receiving a service request *FFSMessage* message, the *ext2* file system server marshals the message and calls the appropriate *ext2* file system functions. In our fos implementation, we have also implemented the block-device-driver as a server which deploys the Xen paravirtualized frontend block-device-driver and interacts with the Xen backend block-device in Dom0. Furthermore, we implemented a Xenbus driver in order to be able to obtain the initialization information provided by Dom0 through the xenbus, and which is needed for the device.

In order to measure the impact of the messaging layer on the fos file system implementation, we measured the time taken by our file system in reading and writing files of different sizes. For that, we have used the default Xen block-device-driver sector size of 512 bytes with fos file system block size of 1024 bytes. We also compare and contrast our performance with the an *ext2* file system executing in a paravirtualized DomU with Linux 2.6.27 OS kernel. We measured our performance results with and without file system caching. On Linux, we used the O_Direct flag to enable and disable caching for *ext2* Linux. Our performance measurement are displayed in Figure 5.

In this experiment, we used a test application which reads and writes a file to the file system in 1KB chunks, and calculate the median of 20 runs. We collected the performance for two file sizes: 1KB and 64KB. In Figure 5, the x-axis represents the total time taken to read or write the file while the y-axis describes the file system operation performed. For each operation, we report on the median fos total time (blue) and the median of Linux DomU total time (green). The upper four bar sets represent reading and writing without file system caching, while the lower four bar sets represent reading and writing with file caching enabled.

We observe that our performance results experience higher variability than Linux, which might be caused by the variable latency of messaging between servers. One of our current optimization focus is to investigate this performance variability. We are also working on performance tuning our file system server and extending it to a parallel file system which leverages the power of the cloud.

5.5 Web Server

The fos web server is a prototypical example of a fos application. The web server listens on port 80 for incoming requests; when it receives a request, it serves a fixed static page. It fetches this page by requesting it directly from the block device; neither the web server nor the block device server cache the page.

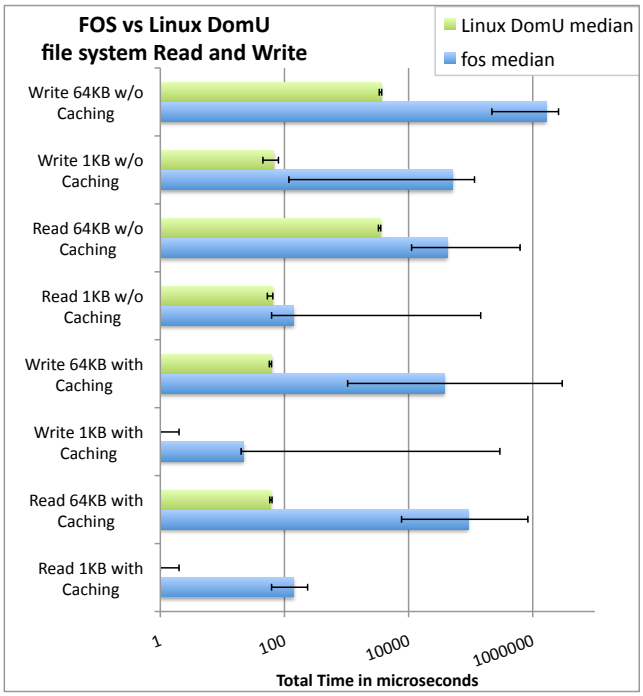

Figure 5: Overall latency experienced by the fos and Linux DomU *ext2* file systems in reading and writing files of varying sizes from the file system.

We measured the values in table 5 using ApacheBench [3]. We ran 1000 non-concurrent requests and measured the average response time. We ran this test 25 times, and found the minimum, average, maximum, and standard deviation of these average response times.

We wrote a similar webserver for Linux, and forced it to read from the disk on every request by opening the file with O_DIRECT. We ran this http server in a domU Linux instance in the same setup, and used ApacheBench to find response times in the same way.

5.6 Single System Image Growth

We used the fos proxy server and cloud interface server to extend an already running fos OS instance. For this test, we used Eucalyptus on our cluster as the cloud manager. The fos cloud interface server used the EC2 REST API to communicate with the Eucalyptus cloud manager over the network. In this test, a fos VM was manually started, which then started a second fos instance via Eucalyptus. The proxy servers on the two VMs then connected and shared state, providing a single system image by allowing fos native messages to occur over the network. The amount of time it took for the first VM to spawn and integrate the second VM was 72.45 seconds.

This time entails many steps outside of the control of fos including response time of the Eucalyptus cloud controller, time to setup the VM on a different machine with a 2GB disk file, time for the second fos VM to receive a IP via DHCP, and the round trip time of the TCP messages sent by the proxy servers when sharing state. For a point of reference, in [20, 19], the Eucalyptus team found that it takes approximately 24 seconds to start up a VM using eucalyptus, but this is using a very different machine and network setup, making these numbers difficult to compare. As future research, we are interested in reducing the time it takes to shrink and grow multi-

VM fos single system image OSes by reducing many of the outside system effects. In addition, we believe that by keeping a pool of hot-spare fos servers, we can significantly reduce the time it takes to grow and shrink a fos cloud.

6. RELATED WORK

There are several classes of systems which have similarities to fos: traditional microkernels, distributed OSes, and cloud computing infrastructure.

Traditional microkernels include Mach [4] and L4 [16]. fos is designed as a microkernel and extends the microkernel design ideas. However, 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. fos also exploits the "spatialness" of massively multicore processors by spatially distributing servers which provide a common OS function.

Like Tornado [14] and K42 [6], fos explores how to parallelize microkernel-based OS data structures. They are differentiated from fos in that they require SMP and NUMA shared memory machines instead of loosely coupled single-chip massively multicore machines and clouds of multicores. Also, fos targets a much larger scale of machine than Tornado/K42. The recent Corey [9] OS shares the spatial awareness aspect of fos, but does not address parallelization within a system server and focuses on smaller configuration systems. Also, fos is tackling many of the same problems as Barrelfish [8] but fos is focusing more on how to parallelize the system servers as well as addresses the scalability on chip and in the cloud.

fos bears much similarity to distributed OSes such as Amoeba [24], Sprite [21], and Clouds [10]. One major difference is that fos communication costs are much lower when executing on a single massive multicore, and the communication reliability is much higher. Also, when fos is executing on the cloud, the trust model and fault model is different than previous distributed OSes where much of the computation took place on student's desktop machines.

fos differs from existing cloud computing solutions in several aspects. Cloud (*IaaS*) systems, such as Amazon's Elastic compute cloud (EC2) [1], provide computing resources in the form of virtual machine (VM) instances and Linux kernel images. fos builds on top of these virtual machines to provide a single system image across an IaaS system. With the traditional VM approach, applications have poor control over the co-location of the communicating applications/VMs. Furthermore, IaaS systems do not provide a uniform programming model for communication or allocation of resources. Cloud aggregators such as RightScale [22] provide automatic cloud management and load balancing tools, but they are application-specific, whereas fos provides these features in an application agnostic manner. Platform as a service (*PaaS*) systems, such as Google AppEngine [15] and MS Azure [17], represent another cloud layer which provides APIs that applications can be developed on. PaaS systems often provide automatic scale up/down and fault-tolerance as features, but are typically language-specific. fos tries to provide all these benefits in an application- and language- agnostic manner.

7. CONCLUSION

Cloud computing and multicores have created new classes of platforms for application development; however, they come with many challenges as well. New issues arise with fractured resource pools in clouds as well needing to deal with dynamic underlying computing infrastructure due to varying application demand, faults, or energy constraints. Our system, fos, seeks to surmount these issues by presenting a single system interface to the user and by providing a programming model that allows OS system services to scale with demand. By placing key mechanisms for multicore and cloud management in a unified operating system, resource management and optimization can occur with a global view and at the granularity of processes instead of VMs. fos is scalable and adaptive, thereby allowing the application developer to focus on application-level problem-solving without distractions from the underlying system infrastructure.

8. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2), 2009. http://aws.amazon.com/ec2/.

[2] Cloud hosting products - Rackspace, 2009. http://www.rackspacecloud.com/cloud_hosting_products.

[3] ab - apache http server benchmarking tool, 2010. http://httpd.apache.org/docs/2.0/programs/ab.html.

[4] 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.

[5] 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.

[6] 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.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[8] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[9] 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.

[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] A. Dunkels, L. Woestenberg, K. Mansley, and J. Monoses. lwIP embedded TCP/IP stack. http://savannah.nongnu.org/projects/lwip/, Accessed 2004.

[12] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM.

[13] M. Freedman, E. Freudenthal, D. MaziÃĺres, and D. M. Eres. Democratizing content publication with coral. In *In NSDI*,

pages 239–252, 2004.

[14] 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.

[15] GOOGLE App Engine. http://code.google.com/appengine.

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

[17] Microsoft azure. http://www.microsoft.com/azure.

[18] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, Apr. 1965.

[19] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus : A technical report on an elastic utility computing archietcture linking your programs to useful systems. Technical Report 2008-10, UCSB Computer Science, Aug. 2008.

[20] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of 9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 09)*, Shanghai, China, 2009.

[21] 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.

[22] Rightscale home page. http://www.rightscale.com/.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

[24] 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.

[25] 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.

[26] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[27] 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.