Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2011-012 March 9, 2011

Fleets: Scalable Services in a Factored Operating System

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

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

Fleets: Scalable Services in a Factored Operating System

David Wentzlaff Charles Gruenwald III Nathan Beckmann Adam Belay
Harshad Kasture Kevin Modzelewski Lamia Youseff Jason E. Miller
Anant Agarwal

CSAIL Massachusetts Institute of Technology

ABSTRACT

Current monolithic operating systems are designed for uniprocessor systems, and their architecture reflects this. The rise of multicore and cloud computing is drastically changing the tradeoffs in operating system design. The culture of scarce computational resources is being replaced with one of abundant cores, where spatial layout of processes supplants time multiplexing as the primary scheduling concern. Efforts to parallelize monolithic kernels have been difficult and only marginally successful, and new approaches are needed.

This paper presents *fleets*, a novel way of constructing scalable OS services. With *fleets*, traditional OS services are factored out of the kernel and moved into user space, where they are further parallelized into a distributed set of concurrent, message-passing servers. We evaluate fleets within fos, a new factored operating system designed from the ground up with scalability as the first-order design constraint. This paper details the main design principles of fleets, and how the system architecture of fos enables their construction.

We describe the design and implementation of three critical fleets (network stack, page allocation, and file system) and compare with Linux. These comparisons show that fos achieves superior performance and has better scalability than Linux for large multicores; at 32 cores, fos's page allocator performs $4.5\times$ better than Linux, and fos's network stack performs $2.5\times$ better. Additionally, we demonstrate how fleets can adapt to changing resource demand, and the importance of spatial scheduling for good performance in multicores.

1. INTRODUCTION

Trends in multicore architectures point to an ever-increasing number of cores available on a single chip. Moore's law predicts an exponential increase in integrated circuit density. In the past, this increase in circuit density has translated into higher single-stream performance, but recently single-stream performance has plateaued and industry has turned to adding cores to increase processor performance. In only a few years, multicores have gone from esoteric to commonplace: 12-core single-chip offerings are available from major vendors [3] with research prototypes showing many more cores on the horizon [26, 19], and 64-core chips are available from embedded vendors [30] with 100-core chips available this year [2]. These emerging architectures present new challenges to OS design, particularly in the management of a previously unprecedented number of computational cores.

Given exponential scaling, it will not be long before chips with hundreds of cores are standard, with thousands of cores following close behind. Recent research, though, has demonstrated problems with scaling monolithic OS designs. In monolithic OSs, OS code executes in the kernel on the same core which makes an OS service request. Corey [10] showed that this led to significant performance degradation for important applications compared to intelligent, application-level management of system services. Prior work by Wentzlaff [29] also showed significant cache pollution caused by running OS code on the same core as the application. This becomes an even greater problem if multicore trends lead to smaller per-core cache. Wentzlaff also showed severe scalability problems with OS microbenchmarks, even only up to 16 cores.

A similar, independent trend can be seen in the growth of cloud computing. Rather than consolidating a large number of cores on a single chip, cloud computing consolidates cores within a data center. Current Infrastructure as a Service (IaaS) cloud management solutions require a cloud computing user to manage many virtual machines (VMs). Unfortunately, this presents a fractured and non-uniform view of resources to the programmer. For example, the user needs to manage communication differently depending on whether the communication is within a VM or between VMs. Also, the user of an IaaS system has to worry not only about constructing their application, but also about system concerns such as configuring and managing communicating operating systems. There is much commonality between constructing OSs for clouds and multicores, such as the management of unprecedented number of computational cores and resources, heterogeneity, and possible lack of widespread shared memory.

The primary question facing OS designers over the next ten years will be: *What is the correct design of OS services that will scale up to hundreds or thousands of cores?* We argue that the structure of monolithic OSs is fundamentally limited in how they can address this problem.

fos [31] is a new factored operating system (OS) designed for future multicores and cloud computers. In contrast to monolithic OS kernels, the structure of fos brings scalability concerns to the forefront by decomposing an OS into services, and then parallelizing within each service. To facilitate the conscious consideration of scalability, fos system services are moved into userspace and connected via messaging. In fos, a set of cooperating servers which implement a single system service is called a *fleet*. This paper describes the implementation of several fleets in fos, the design principles used in their construction, and the system architecture that makes it possible.

Monolithic operating systems are designed assuming that computation is the limiting resource. However, the advent of multicore and clouds is providing abundant computational resources and changing the tradeoffs in OS design. New challenges have been introduced through heterogeneity of communication costs and the unprecedented scale of resources under management. In order to address these challenges, the OS must take into account the spatial layout of processes in the system and efficiently manage data and resource sharing. Furthermore, these abundant resources present opportunities to the OS to allocate computational resources to auxiliary purposes, which accelerate application execution, but do not run the application itself. fos leverages these insights by factoring

OS code out of the kernel and running it on cores separate from application code.

As programmer effort shifts from maximizing per-core performance to producing parallel systems, the OS must shift to assist the programmer. The goal of fos is to provide a system architecture that enables and encourages the design of fleets. In this paper, we make the following contributions:

- We present the design principles used in the construction of fleets: fleets are *scalable*, and designed with scalability foremost in mind; fleets are *self-aware*, and adapt their behavior to meet a changing environment; and fleets are *elastic*, and grow or shrink to match demand.
- We present the design of three critical fleets in fos (network stack, page allocator, and file system). Additionally, we present the first scaling numbers for these services, showing that fos achieves superior performance compared to Linux on large multicores or when considering application parallelism.
- We present studies of self-aware, elastic fleets, using a prototype file system fleet. These studies demonstrate the importance of good spatial scheduling in multicores and the ability of fleets to scale to meet demand.

This paper is organized as follows. Section 2 presents fos's architecture. Section 3 presents the design principles of fleets. Section 4 presents the design of several important fleets. Section 5 presents a case study of fos scaling across the cloud, and exercises the full system with a real application. Section 6 presents our results. Section 7 relates fos to previous work, and Section 8 concludes.

2. SYSTEM ARCHITECTURE

Current OSs were designed in an era when computation was a limited resource. With the expected exponential increase in number of cores, the landscape has fundamentally changed. The question is no longer how to cope with limited resources, but rather how to make the most of the abundant computation available. fos is designed with this in mind, and takes scalability and adaptability as *the* first-order design constraints. The goal of fos is to design system services that scale from a few to thousands of cores.

fos does this by factoring OS services into userspace processes, running on separate cores from the application. Traditional monolithic OSs time multiplex the OS and application, whereas fos spatially multiplexes OS services (running as user processes) and application processes.[1] In a regime of one to a few cores, time multiplexing is an obvious win because processor time is precious and communication costs are low. With large multicores and the cloud, however, processors are relatively abundant and communication costs begin to dominate. Running the OS on every core introduces unnecessary sharing of OS data and associated communication overheads; consolidating the OS to a few cores eliminates this. For applications that do not scale well to all available cores, factoring the OS is advantageous in order to accelerate the application. In this scenario, spatial scheduling (layout) becomes more important than time multiplexing within a single core.

However, even when the application could consume all cores to good purpose, running the OS on separate cores from the application provides a number of advantages. Cache pollution from the OS is reduced, and OS data is kept hot in the cache of those cores running the service. The OS and the application can run in parallel, pipelining OS and application processing, and often eliminating expensive context switches. Running services as independent threads

[1]While spatial multiplexing is the primary scheduling medium in fos, it also supports time multiplexing.

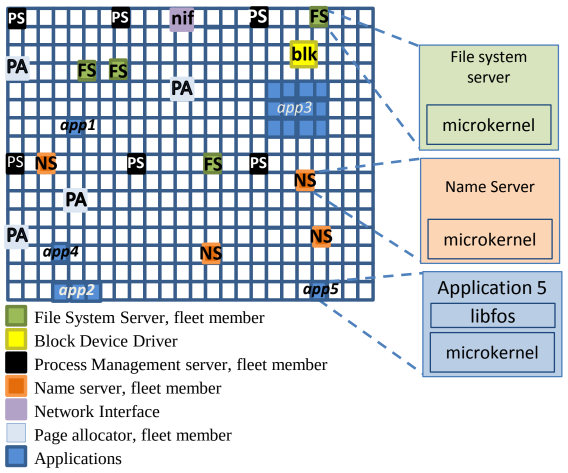


Figure 1: A high-level illustration of the fos servers layout in a multicore machine. This figure demonstrates that each OS service fleet consists of several servers which are assigned to different cores.

of execution also enables extensive background optimizations and re-balancing. Although background operations exist in monolithic OSes,[2] fos facilitates such behavior since each service has its own thread of control.

In order to meet demand in a large multicore or cloud environment, reduce access latency to OS services and increase throughput, it is necessary to further parallelize each service into a set of distributed, cooperating servers. We term such a service a *fleet*.

Figure 1 shows the high-level architecture of fos. A small microkernel runs on every core. Operating system services and applications run on distinct cores. Applications can use shared memory, but OS services communicate only via message passing. A library layer (*libfos*) translates traditional syscalls into messages to fos services. A naming service is used to find a message's destination server. The naming service is maintained by a fleet of naming servers. Finally, fos can run on top of a hypervisor and seamlessly span multiple machines, thereby providing a single system image across a cloud computer. The following subsections describe the architecture of fos, and the next section covers how fos supports building fleets.

2.1 Microkernel

In order to factor OS services into fleets, fos uses a minimal microkernel design. The microkernel provides only: (i) a protected messaging layer, (ii) a name cache to accelerate message delivery, (iii) rudimentary time multiplexing of cores, and (iv) 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. However, many OS system services possess special capabilities that grant them privileges beyond those of regular applications.

Capabilities are extensively used to restrict access into the protected microkernel. For instance, the memory modification API allows 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 processes. Capabilities are also used in the messaging system to determine who is allowed to send messages to whom.

2.2 Messaging

[2]*E.g.,* kernel threads such as `kswapd` and `bdflush` in Linux.

fos provides interprocess communication through a mailbox-based message-passing abstraction. The API allows processes to create mailboxes to receive messages, and associate the mailbox with a name and capability. This design provides several advantages for a scalable OS on multicores and in the cloud. Messaging can be implemented via a variety of underlying mechanisms: shared memory, hardware message passing, TCP/IP, *etc.*. This allows fos to run on a variety of architectures and environments.

The traditional alternative to message-passing is shared memory. However, in many cases shared memory may be unavailable or inefficient: fos is architected to support unconventional architectures [18, 19] where shared memory support is either absent or inefficient, as well as supporting future multicores with thousands of cores where global shared memory may prove unscalable. Relying on messaging is even more important in the cloud, where machines can potentially reside in different datacenters and intermachine shared memory is unavailable.

A more subtle advantage of message passing is the programming model. Although perhaps less familiar to the programmer, a message-passing programming model makes data sharing more explicit. This allows the programmer to consider carefully the data sharing patterns and find performance bottlenecks early on. This leads to more efficient and scalable designs. Through message passing, we achieve better encapsulation as well as scalability.

It bears noting that fos supports conventional multithreaded applications with shared memory, where hardware supports it. This is in order to support legacy code as well as a variety of programming models. However, OS services are implemented strictly using messages.

Having the OS provide a single message-passing abstraction allows transparent scale-out of the system, since the system can decide where best to place processes without concern for straddling shared memory domains as occurs in cloud systems. Also, the flat communication medium allows the OS to perform targeted optimizations across all active processes, such as placing heavily communicating processes near each other.

fos currently provides three different mechanisms for message delivery: kernelspace, userspace, and intermachine. These mechanisms are transparently multiplexed in the *libfos* library layer, based on the locations of the processes and communication patterns:

Kernelspace The fos microkernel provides a simple implementation of the mailbox API over shared memory. This is the default mechanism for delivering messages within a single machine. Mailboxes are created within the address space of the creating process. Messages are sent by trapping into the microkernel, which checks the capability and delivers the message to the mailbox by copying the message data across address spaces into the receiving process. Messages are received without trapping into the microkernel by polling the mailbox's memory. The receiver is not required to copy the message a second time because the microkernel is trusted to not modify a message once it is delivered.

Userspace For processes that communicate often, fos also provides shared memory channel-based messaging inspired by URPC [9] and Barrelfish [8]. The primary advantage of this mechanism is that it avoids system call overhead by running entirely in user space. Channels are created and destroyed dynamically, allowing compatibility with fos's mailbox messaging model. Outgoing channels are bound to names and stored in a user-level name cache. When a channel is established, the microkernel maps a shared page between the sender and receiver. This page is treated as a circular queue of messages. Data must be copied twice, once by the sender when the message is enqueued in the buffer and once by the receiver when the message is dequeued from the buffer. The second copy is needed for security and to ensure the queue slot is available for future messages as soon as possible. As shown later in this paper, this mechanism achieves much better per-message latency, at the cost of an initial overhead to establish a connection.

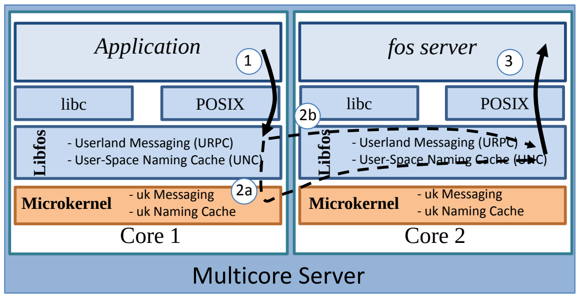


Figure 2: fos messaging over kernel (2a) and userspace (2b).

Intermachine Messages sent between machines go through a proxy server. This server is responsible for routing the message to the correct machine within the fos system, encapsulating messages in TCP/IP, as well as maintaining the appropriate connections and state with other proxy servers in the system.

Figure 2 shows the two intra-machine messaging transports. Messages from an application are sent through *libfos*, which chooses to send them via kernel (2a) or userspace (2b) messaging. The *libfos* on the receiving side then delivers the data to the fos server.

It is important to note that these mechanisms are completely transparent to the programmer. As messaging is fundamental to fos performance, making intelligent choices about which mechanism to use is critical.

2.3 Name Service

Closely coupled with messaging, fos provides a name service to lookup mailboxes throughout the system. Each name is a hierarchical URI much like a web address or filename. The namespace is populated by processes registering their mailboxes with the name service. The key advantage of the name service is the level of indirection between the symbolic identifier of a mailbox and its so-called "address" or actual location (machine, memory address, *etc.*). By dealing with names instead of addresses, the system can dynamically load balance as well as re-route messages to facilitate and processes migration.

The need for dynamic load balancing and process migration is a direct consequence of the massive scale of current cloud systems and future multicores. In addition to a greater amount of resources under management, there is also greater variability of demand. Static scheduling is inadequate, as even if demand is known it is rarely constant. It is, therefore, necessary to adapt the layout of processes in the system to respond to where the service is currently needed.

The advantage of naming is closely tied to fleets. Fleet members will each have an in-bound mailbox upon which they receive requests, and these mailboxes will all be registered under a single name. It is the responsibility of the name service to resolve a request to one member of a fleet.

Load balancing can be quite complicated and highly customized to a specific service. Each service can dynamically update the name system to control the load balancing policy for their fleet. The name service does not *determine* load balancing policy, but merely provides mechanisms to *implement* a policy. To support stateful operations, applications or *libfos* can cache the name lookups so that all messages for a transaction go the same fleet member.[3] Alternatively, the fleet can manage shared state so that all members can handle any request.

In fos, another design point is to explicitly load balance within the fleet. This approach may be suitable when the routing decision is based on state information not available to the name service. In either approach it is important to realize that by decoupling the lookup of mailboxes using a symbolic name, the OS has the freedom to implement a given service through a dynamic number of servers.

The name service also enables migration of processes and their mailboxes. This is desirable for a number of reasons, chiefly to improve performance by moving communicating servers closer to each other. Migration is also useful to rebalance load as resources are freed. The name service provides the essential level of indirection that lets mailboxes move freely without interrupting communication.

3. FLEETS

The previous section covered the system architecture components of fos. This section discusses how fos supports building fleets, and the principles used in building them. The programming model used to construct fleets is also discussed, highlighting the tools and libraries provided by fos to ease their construction.

3.1 Overview

Services in fos are implemented by cooperating, spatially-distributed sets of processes. This idea is the cornerstone of fos. Whereas prior projects have demonstrated the viability of microkernels, fos aims to implement a complete distributed, parallel OS by implementing service fleets. The core design principles of fleets are:

- *Scalability.* Fleets are designed with scalability as the primary design constraint. Fleets employ best practices for scalability such as lockless design and data partitioning, as well as the best available data structures and algorithms.
- *Self-awareness.* Fleets monitor and adapt their behavior to the executing environment. Load between members is rebalanced, and members are migrated to improve communication latency.
- *Elasticity.* Fleets are elastic, and can expand and shrink to match changing demand. Performance is monitored such that the optimal number of servers is used to implement each OS service.

Each system service is implemented by a single fleet of servers. Within a single system, there will be a file system fleet, a page allocator fleet, a naming fleet, a process management fleet, *etc.*. Additionally, the fleet may span multiple machines where advantageous. For example, in order to provide local caching for fast access, it is good practice to have a member of the file system fleet on every machine. The same general principle applies to many OS services, and for some critical services (*viz.*, naming) it is required to have an instance on each machine.

Fleets must support a variety of management tasks. Fleets can grow and shrink to meet demand, and must support rebalancing when a new member joins or leaves the fleet. Currently many services designate a single member, termed the *coordinator*, to perform many of these tasks.

[3]For example, the name lookup of `/foo/bar` results in the symbolic name `/foo/bar/3`, which is the third member of the fleet. This is the name that is cached, and subsequent requests forward to this name, wherever it should be.

3.2 Programming Model

fos provides libraries and tools to ease the construction of fleets and parallel applications. These are designed to mitigate the complexity and unfamiliarity of the message-passing programming paradigm, thus allowing efficient servers to be written with simple, straight-line code. These tools are (i) a cooperative threading model integrated with fos's messaging system, (ii) a remote procedure call (RPC) code generation tool, and (iii) a library of distributed objects to manage shared state.

The cooperative threading model and RPC generation tool are similar to tools commonly found in other OSs. The cooperative threading model lets several active contexts multiplex within a single process. The most significant feature of the threading model is how it is integrated with fos's messaging system. The threading model provides a dispatcher, which implements a callback mechanism based on message types. When a message of a particular type arrives, a new thread is spawned to handle that message. Threads can send messages via the dispatcher, which sleeps the thread until a response arrives. The use of a cooperative threading model allows the fleet server writer to write straight-line code for a single transaction and not have to worry about preemptive modification of data structures thereby reducing the need for locks.

The RPC code generator provides the illusion of local function calls for services implemented in other processes. It parses regular C header files and generates server and client-side libraries that marshall parameters between servers. The tool parses standard C, with some custom annotations via gccxml [4] indicating the semantics of each parameter. Additionally, custom serialization and deserialization routines can be supplied to handle arbitrary data structures. The RPC tool is designed on top of the dispatcher, so that servers implicitly sleep on a RPC call to another process until it returns.

The libraries generated by the RPC tool provide more general support for constructing fleets. For example, they can be used to pack and unpack messages without RPC (send, sleep, return) semantics. This is useful in order to pipeline requests with additional processing, broadcast a message to fleet members, and support unusual communication patterns that arise in constructing fundamental OS services.

One challenge to implementing OS services as fleets is the management of shared state. This is the major issue that breaks the illusion of straight-line code from the RPC tool. fos addresses this by providing a library of distributed data structures that provide the illusion of local data access for distributed, shared state.

The goal of this library is to provide an easy way to distribute state while maintaining performance and consistency guarantees. Data structures are provided matching common usage patterns seen in the implementation of fos services. The name service provides a distributed key-value store, implemented via a two-phase commit protocol with full replication. The library provides another key-value store implementation that distributes the state among participants. These implementations have different cost models, and usage dictates when each is appropriate. Similarly, the page allocator uses a distributed buddy allocator; this data structure could be leveraged to provide process IDs, file pointers, etc..

These data structures are kept consistent using background updates. This is achieved using the cooperative dispatcher discussed above. The distributed data structure registers a new mailbox with the dispatcher and its own callbacks and message types.

We are currently exploring the set of data structures that should be included in this library, and common paradigms that should be captured to enable custom data structures. Later in the paper, we discuss how each parallel service manages its shared data in detail.

3.3 Design Principles

This section discusses the three core design principles in greater depth including how the fleet architecture facilitates them.

3.3.1 Scalability

Fleets are designed to scale from a few to very many servers. They are not tuned to a particular size, but designed using best practices and algorithms to scale over a large range of sizes. This is important for multicore and cloud computing, as current trends in increasing core counts are likely to continue for the foreseeable future. Furthermore, different processors, even within a single processor family, will have variety of core counts. Therefore, fleets are designed to scale to different number of cores to address these needs.

In order to facilitate the scalability of fos fleets, fleets are designed in a message-passing-only manner such that layout of the data is explicit and shared memory and lock contention do not become bottlenecks. Our results show that lock contention in Linux has major scalability impact on the page allocation service, whereas fos is able to achieve excellent scalability through lockless design.

3.3.2 Self-awareness

A natural advantage of separating OS services from applications is the ease of performing background optimizations and re-balancing of the service. Although such optimizations are possible in monolithic designs, giving each service its own thread provides a natural framework in which to perform such tasks. Interference with application performance can be minimized by performing tasks only when necessary or when the service is idle.

Fleets monitor their environment and adapt their behavior to improve performance. For example, fleet members can migrate to minimize communication costs with cores they are serving. Similarly, when a new transaction begins, it is assigned to the closest available fleet member. Active transactions can be migrated to other members if a server becomes overloaded, and these performance statistics also motivate growing or shrinking the fleet.

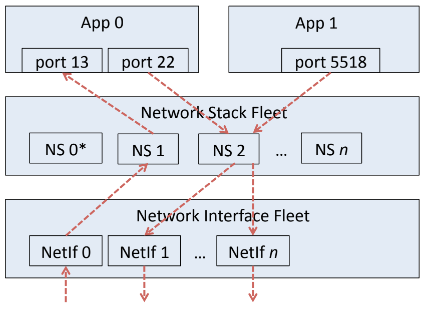
Fleets often must route requests themselves, independent of the name service. One important reason is resource affinity – if a request uses a resource under management of a particular fleet member, then the request should be forwarded to that member. A simple example of this is local state kept by each fleet member for a transaction, for example a TCP/IP connection. In this case, routing through the name service is insufficient because state has already been created during connection establishment, and the connection is associated with a particular fleet member when the first message on that connection arrives (see Section 4.1). Another example is if a request uses a hardware resource on a different machine. In this case, the request must be forwarded to the fleet member on the machine that has access to the hardware.

3.3.3 Elasticity

In addition to unprecedented amounts of resources, clouds and multicores introduce unprecedented variability in demand for these resources. Dynamic load balancing and migration of processes go a long way towards solving this problem, but still require over-provisioning of resources to meet demand. This would quickly become infeasible, as every service in the system claims the maximum amount of resources it will ever need. Instead, fleets are *elastic*, meaning they can grow to meet increases in demand, and then shrink to free resources back to the OS.

Monolithic OSs achieve elasticity by "accident", as OS code runs on the same core as the application code. This design has obvious advantages, since the computational resources devoted to the service scale proportionally with demand. There are disadvantages, however: monolithic designs relinquish control of how many cores to provision the service. This can lead to performance degradation if too many threads are accessing a shared resource simultaneously. fos can avoid this by fixing the size of a fleet at the point that achieves maximal performance. One example of "elasticity by accident" running awry occurs when a single lock is highly contended. In this case, when more cores contend for a lock, the performance of all cores degrades. Limiting the numbers of cores performing OS functions (contending for the resource) can actually improve performance in such cases. Our results show examples of this phenomenon where by limiting the number of cores dedicated to a fleet, fos can achieve higher performance with fewer resources than Linux simply because Linux has no means to limit the number of cores running the OS services. Additionally, for applications that rely heavily on the OS it may be best to provision *more* cores to the OS service than the application. The servers can then collaborate to provide the service more efficiently. These design points are not provided in monolithic operating systems.

A fleet is grown by starting a new server instance on a new core. This instance joins the fleet by contacting other members (either the coordinator or individual members via a distributed discovery protocol) and synchronizing its state. Some of the distributed, shared state is migrated to the new member, along with the associated transactions. Transactions are migrated in any number of ways, for example by sending a redirect message to the client from the "old" server. Shrinking the fleet can be accomplished in a similar manner.



* Netstack 0 acts as the coordinator.

Figure 3: Logical diagram of network communication components

4. FOS FLEET IMPLEMENTATIONS

This section presents parallel implementations of several system services in fos. We present four fos services: a TCP/IP network stack, a page allocator, a process management service, and a prototype file system.

4.1 Network Stack

fos has a fully-featured networking service responsible for packing and unpacking data for the various layers of the network stack as well as updating state information and tables associated with the various protocols (*e.g.*, DHCP, ARP, and DNS). The stack was implemented by extending lwIP [14] with fos primitives for parallelization to create the network stack fleet. The logical view of this service is depicted in Figure 3. In this diagram the dashed lines represent the paths that a given TCP/IP flow may take while traversing the network stack. In this diagram we can see that the flows are multiplexed between the different network stack fleet members. The distribution of these flows amongst the fleet members is managed by the fleet coordinator.

The design employs a fleet of network stack servers with a single

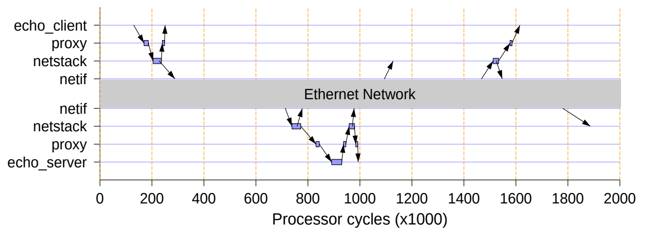


Figure 4: Echo client and echo server communicating via the networking service

member designated as the coordinator. The fleet coordinator is responsible for several management tasks as well as handling several of the protocols.

When the kernel receives data from the network interface it delivers it to the network interface server. The network interface server then peeks into the packet and delivers it to one of the fleet members depending on the protocol the packet corresponds to. The handling of many stateless protocols (UDP, ICMP) is fairly straightforward, as they can be passed to any member. Likewise, low-frequency stateful requests (DNS, DHCP, ARP) can be handled by a single fleet member, broadcasting information required to all fleet members. Therefore, the remainder of this section discusses TCP, which is the dominant workload of the network stack and exposes the most challenging problems.

Since TCP flows are stateful they must be handled specially, demonstrating how fleet members can coordinate to handle a given OS service. When an application wishes to listen on a port it sends a message to the coordinator which adds state information associated with that application and port. Once a connection has been established, the coordinator passes responsibility for this flow to a fleet member. The coordinator also sets up a forwarding notification such that other packets destined for this flow already in the coordinator's queue get sent to the fleet member who is assigned this flow. While this approach potentially re-orders packets, as the forwarded packets can be interleaved with new input packets, TCP properly handles any re-ordering. Once the fleet member has accepted the stream, it notifies the network interface to forward flows based on a hash of the *(source IP, source port, destination IP, destination port)*. Once this flow forwarding information has been updated in the network interface server, packets of this type are delivered directly to the fleet member and then the application. Note that these mechanisms occur behind *libfos* and are abstracted from the application behind convenient interfaces.

Figure 4 shows a real network trace from fos of an echo client and echo server communicating via the network stack. Each arrow represents a message between servers. The request from the client is sent out over Ethernet via the proxy server, network stack and network interface; the networking service on the server receives this message, passes it on to the echo server which sends a reply back to the echo client.

4.2 Page Allocator

The page allocator service is responsible for managing all the available physical memory in the system. Requests for physical pages, whether from a user application or from another system service, are handled by the page allocator, as shown in Figure 5.

The page allocator service is implemented as a fleet of identical servers which together implement a distributed buddy allocator. Each server in the fleet is capable of servicing requests for the allocation or freeing of pages. In addition, the page allocator service also exposes an allocate-and-map functionality where the

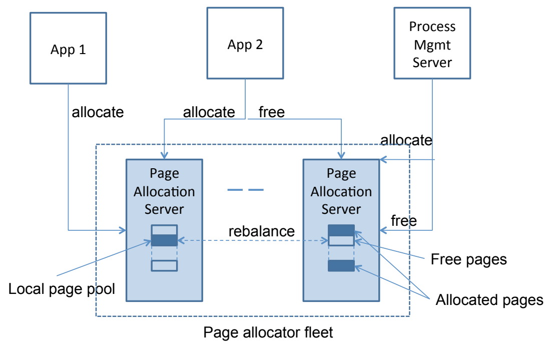


Figure 5: The page allocator fleet handling requests for physical memory from other system services and user applications

allocated pages are also mapped in the requester's address space by the server. Fleet members coordinate to ensure that each page is only allocated to one process at a time, and that freed pages become available again for allocation.

Like a typical buddy allocator, the page allocation service only allocates and frees blocks of pages whose size is a power of two. At system start-up, each fleet member is assigned an initial set of pages to manage. When a server receives an allocation request, it tries to service it from its local pool of pages. If this is not possible, the request is forwarded to another server that it believes has enough pages. Requests to free pages can be sent to any server, regardless of which server allocated them (this might be desirable if the server that allocated the pages is congested, has crashed, or has been migrated to a distant location).

When the servers are not busy servicing requests, they perform background optimizations to improve performance. For instance, the distribution of free pages within fleet members is kept uniform via the use of a *low-watermark*. When a server notices that the number of free pages it has has fallen below the *low-watermark*, it requests pages from other servers.

Another background optimization is aimed at mitigating the problem of *distributed fragmentation*. This occurs when there are enough free pages in the system, but they are distributed among various servers and are thus unable to be coalesced. As a result, requests for large contiguous page blocks can not be satisfied. The page allocator service combats this problem by dividing up the page-space into large chunks, and assigning each chunk to a "home server". While pages may be freed back to servers other than their home server, they are periodically rebalanced in the background. This rebalancing is triggered whenever a server hits a *high-watermark* of free pages, causing it to send pages back to their home servers, where they can be coalesced with other free pages. Since this and other optimizations happen in the background, they have minimal impact on the latency experienced by the clients.

4.3 Process Management

The process management service handles a variety of tasks including process creation, migration, and termination, as well as maintaining statistics about process execution. Like other system services in fos, the process management service is implemented as a fleet of servers.

The process management service performs most of the tasks involved in process creation, including setting up page tables and the process context, in user space. Traditional operating systems perform these tasks in kernel space. Pulling this functionality out of the kernel allows fos to significantly shrink the size of the kernel code base, and allows the implementation of a scalable process creation mechanism using the same techniques used for writing other scalable system services in fos. The role of the kernel is reduced to simply validating the changes made by the process management service; this validation is done via capabilities.

A process creation request from a user process is forwarded to one of the process management servers. The server then queries the file system to retrieve the executable to be spawned. The physical memory needed for the page tables as well as the code and data in the spawned process's address space is allocated via the page allocation server. Writing to this memory requires special capabilities which need to be presented to the microkernel. Finally, the service sets up the new process' context and adds the process to the scheduler's ready queue. This operation is also guarded by a microkernel capability.

Because process creations are handled by a separate set of servers, fos implements asynchronous process creation semantics, where a process can fire off multiple process creation requests and continue doing useful work while these processes are being set up. It can then check for a notification from the process management service when it needs to communicate with the newly spawned processes.

The service also collects and maintains statistics during process execution, such as resource utilization and communication patterns. These statistics can then be made available to other system services such as the scheduler.

Lastly, the service also handles process termination, including faults encountered by the process. Upon process termination, resources used by the terminating process (*e.g.*, memory) are freed and the state maintained within the process management server is updated to reflect this.

4.4 File System

fos implements a prototype file system fleet based on the ext2 file system. The file system fleet is supported by the block device server, which manages disk I/O. Application interaction is handled by *libfos*, which intercepts POSIX file system calls and generates messages to the file system fleet. In response to these requests, the file system interacts with the block device server to retrieve the data, and responds via a message to the application when the response is ready. Each file system server communicates with the block device through *libfos*, which provides caching of blocks. This limits traffic to the block device service, preventing it from becoming a bottleneck for repeated requests.

fos contains a parallel file system fleet (currently read-only) that consists of several servers interacting with the same block device. This implementation requires no data sharing between fleet members, but for correctness all requests to an open file must go to the same file system server. This is achieved by *libfos* caching the filesystem name lookup while any files are open. If no files are open when a request is made, then the cache is refreshed and a new (possibly different) fleet member is contacted to serve the request.

The primary purpose of the file system is to demonstrate the self-aware and elastic abilities of fos fleets. The file system is self-aware, and monitors its utilization to determine when to grow and shrink. Utilization on a per-fleet-member basis is determined by checking if new requests are available on its input mailbox when the service is idle. The time spent waiting for messages and the time spent processing requests yields utilization. The global utilization of the fleet is then determined by passing utilization messages around the fleet in a ring fashion. When the fleet coordinator has determined that the utilization is above or below desired thresholds the size of the fleet is adjusted accordingly.

5. CASE STUDY: FOS IN THE CLOUD

A unique capability of fos is its ability to use more resources than can be contained within a single machine. In order to achieve this, fos launches additional fos VM instances on different machines. These new VMs natively join the currently running fos instances and the building blocks of fos, naming and messaging, are transparently bridged via the proxy server. Previous work [31] describes how a proxy server is used to transparently build a single system image across a cloud.

To demonstrate fos's cloud functionality, its ability to scale across a cloud computer and as a full system test, we have constructed an application which mimics the key parts of a website such as YouTube. This test exercises the network stack, the file system, and the messaging system (including proxy service) in a large-scale application. In this test, a user accesses a webpage served by the fos webserver; in response to this request, the webserver launches copies of ffmpeg 1.5.0 to transcode a video into two different file formats. Finally the user downloads the files from the webserver. For this test, we used fos to transcoded the first YouTube video, "Me at the zoo", from Flash Video (H.263) to two output formats - an AVI encoded in Windows Media 8, and an AVI encoded in MPEG2. Each video is between 600KB - 700KB.

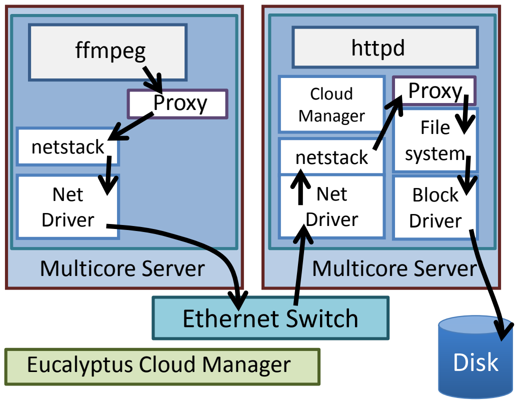


Figure 6: Servers used in the video transcode application. Arrows show messages sent while ffmpeg accesses remote file system.

Figure 6 shows the key components of this test. The test begins by spawning the fos VM on the right. The Cloud Manager Server in the right fos VM contacts the Eucalyptus Cloud infrastructure to start a second fos VM on a different machine (the fos VM on the left in Figure 6). This VM joins the currently running fos instance and messaging and naming are transparently bridged via the Proxy Servers. Next a user accesses a CGI page hosted on the webserver to begin the transcoding process. The httpd server messages the left fos VM which launches ffmpeg to transcode the input FLV file into two AVI outputs. The fos VM containing ffmpeg does not have a file system attached therefore all of the file system access are proxied over the network between machines to the right fos VM. After the transcoding is complete, ffmpeg writes the result files to the file system and client downloads the transcoded video from the httpd server. This whole transaction takes 4:45 minutes with the bulk of the time spent in the transcode followed by the time taken to serve the final transcoded files. It should be noted that due to the transparency at the messaging layer, no code in either ffmpeg or any associated libraries needed to be changed to do inter machine file system access, thus demonstrating the power and utility of fos's single system image across the cloud.

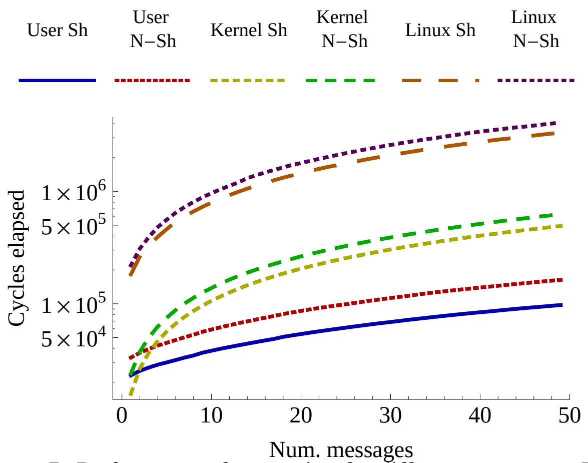


Figure 7: Performance of messaging for different transports. Results are included for cores with a shared cache ("*Sh*") and without ("*N–Sh*"). For reference, UNIX domain sockets in Linux are also included ("*Linux Sh*" and "*Linux N–Sh*").

6. EVALUATION

This section presents results for the fleets described above. Additionally, we provide results and analysis for several of the underlying mechanisms that are critical to system performance. The methodology is first described. Performance of messaging is shown, including tradeoffs between different schemes and their performance bottlenecks. Then scaling numbers for each OS service are presented using microbenchmarks, with comparison against Linux included. We conclude with two studies using the file system: a study on the impact of spatial layout, and a study of elasticity.

6.1 Methodology

fos runs as a paravirtualized OS under Xen 4.0 [7]. This is done in order to facilitate fos's goal of running across the cloud, as the cloud management platforms Amazon EC2 and Eucalyptus both exclusively support Xen domUs. The use of Xen also reduces the number of hardware-specific drivers that needed to be written, which is a challenge for any experimental OS. The ideas used in fos as well as the fleet design presented here are independent of running on a hypervisor, and we plan to construct a bare-metal implementation of fos in the future. All numbers presented in this paper, for both fos and Linux, are presented running in Xen domU's. A stock Ubuntu 9.04 DomU is used to gather Linux numbers. fos instances run with all cores under a single VM. Our current Xen configuration limits VM instances to 32 VCPUs, which limits the scale of some studies. Dom0 is running a custom Linux kernel based on Debian Lenny.

For scalability results, numbers are presented as service throughput. This throughput is calculated based on the median of service latency over n requests, where n is at least 1000 requests.

This data is gathered on currently available multicores. As fos separates OS and application code, this limits the scale of studies that are able to be performed. Numbers are gathered on two machine configurations: (i) a 48-core (quad-12 core) machine with 1.8 GHz AMD Opteron 6168 processors and 64 GB of RAM, and (ii) a 16-core (quad-quad core) machine with 2.3 GHz Intel Xeon E7340 processors and 16 GB of RAM. Unless otherwise mentioned, assume configuration (i) is used for all studies.

6.2 Messaging

In this study, the performance of kernelspace and userspace messaging on both the Intel and the AMD system is evaluated. In each

		AMD Opteron 6168	Intel Xeon E7340
fos Userspace	shared cache	1311 (279)	858 (216)
	no shared cache	1481 (452)	1278 (297)
fos Kernelspace	shared cache	9745 (1288)	6594 (432)
	no shared cache	10303 (1410)	8999 (723)
DomU Linux syscall		874 (120)	573 (82)

Table 1: A latency comparison of null fos system calls (roundtrip messaging) and null Linux system calls. Measurements are in nanoseconds (σ).

case, latency is measured as the round-trip messaging time between a client and a fos system service — This can be considered a null system call in fos because messaging is frequently used as an alternative to operations that monolithic OSs would normally perform by trapping into the kernel. Table 1 compares the steady-state latency of a null Linux system call (in a Xen DomU) with a null fos system call.

The cumulative latency of a series of messages on the Intel machine is also explored, as shown in Figure 7. Measurements of standard Unix domain sockets (as implemented in Linux) are provided for reference. Setup costs are included with the first message measurement, showing overhead for channel creation and other types of initialization.

For all tests, two spatial placements are explored. This first is to schedule communicating processes on the same die with a shared cache. The second is to schedule communicating processes on separate physical packages and thus without shared cache resources. Heterogeneity in messaging latency is observed, especially on the Intel machine.

6.3 Fleet Microbenchmarks

This section presents microbenchmark results for each parallel OS service. Results are first shown as throughput plotted versus the number of clients (*i.e.*, application cores) making requests to the OS service. This shows the scaling of each service and how scaling is affected by allocating additional resources to the service.

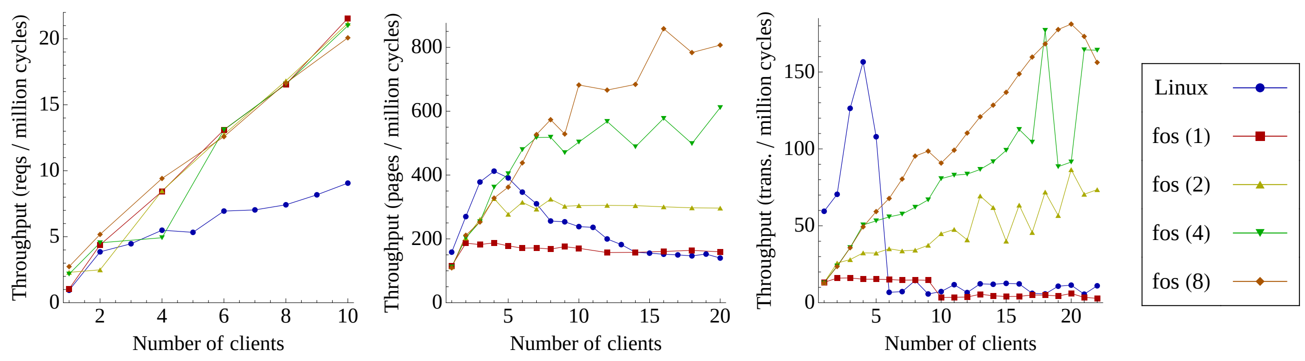
6.3.1 Network Stack

This test compares the latency experienced by small sized packets through the network stack in both Linux and fos for various numbers of streams. The experiment consists of several echo servers running in domU that receive a packet and provide an immediate response, and one client for each flow in dom0 that continuously performs requests. Both the clients and the fos VM were run on the same machine to avoid network latency. In fos, the number of servers in the network stack fleet is varied.

Figure 8a presents our results. The figure plots the number of fleet clients on the horizontal axis and system throughput on the vertical axis. Results are presented for fos using 1-8 network stack servers as well as for Linux. As depicted in the graph, when the number of clients increases the throughput does not scale proportionally for Linux. fos is able to achieve near ideal scaling with a single network stack. This demonstrates that for this particular communication pattern the number of members of the fleet can be quite low. We believe this performance is achieved through reduced cache pollution and because the network stack can pipeline processing with application responses.

6.3.2 Page Allocator

Figure 8b shows the throughput and scalability of the page allocator service for different numbers of clients and servers. Results



(a) Network stack (b) Page allocator. (c) File system.

Figure 8: Performance and scalability of parallel services vs. number of clients making requests to the service. Results are shown for Linux and fos with differently-sized service fleets. fos (n) designates fos running with n servers in the service fleet.

	fos (Sync)	fos (Async)	Linux (F+E)	Linux (F)
Latency (ms)	108.55	0.11	1.17	0.21

Table 2: Process creation latencies in fos and Linux

are collected for Linux by running multiple, independent copies of a memory microbenchmark that allocates pages as quickly as possible, touching the first byte of each to guarantee that a page is actually allocated and mapped. Results are collected from fos by running multiple client processes and a page allocator fleet comprised of multiple servers. Again, the client processes allocate pages as quickly as possible and no lazy allocation occurs.

As can be seen from the figure, the page allocator service with one server in the fleet quickly reaches a bottleneck in the speed at which that server can map pages into the clients' page tables. Increasing the number of servers increases the maximum throughput achievable, allowing the system to satisfy a higher number of clients.

Linux exhibits similar behavior to fos for small numbers of cores, however its throughput peaks (at 4 cores) and then decreases steadily as more clients are added. fos on the other hand shows a continual increase in throughput as the number of clients is increased for higher page allocator core counts. As prior work by Wentzlaff et al [29] indicates, this is a result of increased lock contention. As the number of clients increases, each client tends to spend higher and higher amounts of time contending over locks that protect the central page pool as pages are transferred between it and the core local page pools (with a large chunk of the time being spent in the functions *rmqueue_bulk* and *free_pages_bulk*). As [29] indicates, lock contention soon dominates total execution time, with more than 70% of the time spent contending over locks for 16 clients.

6.3.3 *Process Management*

The semantics of process creation in fos doesn't have a direct Linux analogue, and therefore it is difficult to directly compare process creation latencies. The closest Linux equivalent to a synchronous spawn on fos is a *fork* followed by an *exec*, while the asynchronous spawn semantics is similar to a Linux *fork* (since *fork* employs copy-on-write optimization in creating a clone of the process). Table 2 presents average latencies for these four operations, averaged over 100 operations. Linux outperforms fos on synchronous spawns, mainly because this aspect of fos hasn't been optimized yet. For example, all the pages in the process' address space are loaded into memory at process creation time in fos, while in Linux pages are only loaded on first access, resulting in significant gains in process creation latency. The asynchronous process creation semantics in fos mean that process creation latency can be hidden by doing useful work concurrently in the parent process, as reflected in the significantly lower latency number for asynchronous process creation presented in Table 2.

6.3.4 *File System*

The file system study evaluates the throughput achieved by file systems in Linux and fos. For this study, each client performs 5000 transactions, where each transaction consists of opening a unique file, reading its contents and closing the file. Our parallel file system fleet is a prototype and is currently read-only (we do implement a sequential read-write file system as well). Consequently, in Linux the file system is mounted read-only. Figure 8c presents the overall throughput achieved in each configuration.

The decrease in file system throughput for Linux as the number of clients increases can be attributed to contention over shared data [10]. The fos file system achieves higher throughput than Linux when the number of clients is large. For fos, the achieved throughput flattens for large number of clients when the number of servers is small; the small degradation with a single server as the number of clients is increased can be attributed to contention over the server mailbox. However, higher throughput can be achieved by using more servers in the fleet as shown in Figure 8c.

6.4 Spatial Scheduling

This experiment demonstrates the impact of proper spatial scheduling on performance in multicores. It uses the read-only filesystem fleet with 'good' and 'bad' layouts. Experiments were run on the 16-core Intel Xeon E7340. This machine has very low intra-socket communication cost using userspace messaging, which exposes significant communication heterogeneity (46%) between intra-socket and inter-socket communication.

The filesystem fleet consists of four servers, and the number of clients increases from one to ten (two cores are consumed by the block device and xenbus servers). In the good layout, one file system server resides on each socket. The filesystem component of *libfos* is self-aware, and selects the local filesystem server for all requests. In the bad layout, all servers reside on a single socket and clients are distributed among remaining sockets (forcing inter-socket communication for each request).

Results are shown in Figure 9. The good layout is uniformly better than the bad layout, and averages around 10-20% better performance. There are some anomalous peaks where performance goes to 40% or 97% better for the good layout, and these data are consistent between runs. As can be seen on the figure, these data

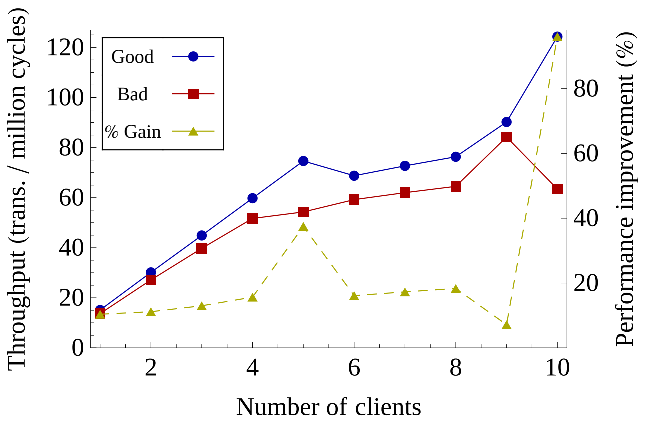


Figure 9: Performance of good vs. bad spatial scheduling. The file system fleet with 4 servers scales from 1 to 10 clients. Figure shows system throughput for good and bad layouts, as well as the performance improvement (dashed) of good vs. bad. The 'good' layout achieves 10-20% better performance on average.

are due to divergent fluctuation in performance at ten clients, which is similar to fluctuations in Figure 8 and we don't believe to be significant. There is also a slight trend towards increasing separation as the number of clients increase.

For large multicores with hundreds of cores, we can expect communication heterogeneity to increase further. Global structures that achieve symmetric performance, *e.g.,* buses, will not scale to large core counts, and new on-chip interconnects will expose heterogeneous communication costs. This trend can already be seen in research prototypes [26, 19] and available multicores [30]. The machine used in this experiment had 46% heterogeneity in communication cost, which led to 10-20% performance difference. Future multicores will feature much greater heterogeneity, and commensurately higher end-to-end performance disparity from spatial scheduling.

6.5 Elasticity

This study focuses on the elastic features of fleets. In this experiment, four clients make queries to the file system service. Each client repeats a single transaction (open a file, read one kilobyte, and close the file) 10,000 times. In order to demonstrate the elastic capabilities of our file system fleet, we constructed the clients to vary their request rates through five different phases. Each phase consists of 2,000 transactions. First, each client request rate is throttled initially to one transaction per two million cycles. The throttling gradually decreases between the 1st and 2,000th transactions. In the second phase, the request rate is kept constant through the phase duration. At the 4,000th transaction, the clients' request rates suddenly increase before gradually decreases again within phase three. The clients' request rates in phase four – between the 6,000th and 8,000th transactions – demonstrate a constant low rate which jumps in phase five to a higher constant request rate.

The file system service is implemented as an elastic fleet. The service starts initially with a single server, which acts as the coordinator. This server monitors the utilization of the service, and detects when the service has become saturated. At this point, the fleet grows by adding a new member to meet demand. Similarly, the coordinator detects if utilization has dropped sufficiently to shrink the fleet, freeing up system resources.

Load balancing is performed through the name server, as discussed previously. Each file system server is registered under a

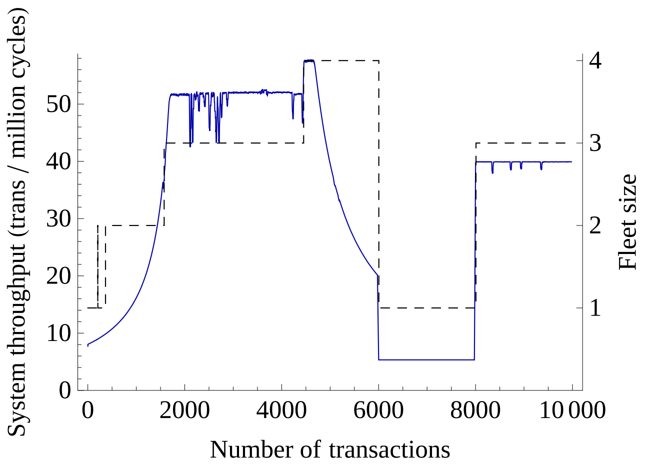


Figure 10: A demonstration of an elastic file system fleet. The number of servers in the fleet is adjusted on the fly to match demand. The solid curve represents the system throughput while the dotted curve represents the fleet-size.

single alias. At the beginning of each transaction, a name look-up is performed through the name server. This name is cached for the remainder of the transaction. The names are served round-robin, performing dynamic load balancing across the different servers in the fleet. This load balancing is performed transparently through *libfos*, by counting the number of open files and only refreshing the cache when no files are open.[4]

Figure 10 shows the results for the elastic fleet of the file system service. Aggregate throughput from the four clients is plotted against the number of transactions in the test. The dotted line represents the size of the file system fleet against the number of transactions in the test. It portrays how the file system fleet responds to the varying clients' request rates by automatically growing and shrinking the number of servers deployed for this service. This configuration briefly saturates a single file system server, before expanding the fleet to two servers. Subsequently, it immediately achieves the throughput of the two server before further expanding the fleet. At the 4,000th transaction, the fleet responds to the sudden increase in the clients' request rates by growing the fleet size to four servers. When the clients request rates again decreases, the fleet shrinks to use fewer resources . Finally, when the request rate again increases in the fifth phase, the fleet deploys more servers to meet the demand.

Figure 10 shows that the elastic fleet throughput follows closely the clients' varying request rates through the different phases of the test, and that there is negligible overhead for performing dynamic load balancing through the name service. The elastic fleet consumes as few cores as needed to meet to the demand for this experiment, only expanding between the "grow" and "shrink" actions when the demand changes.

6.6 Summary

fos can exceed Linux performance and scalability, even when accounting for dedicated OS cores. Figure 11 shows the performance of the page allocator microbenchmark when accounting for the total number of cores consumed, including those dedicated to OS services in fos. Results were similar for omitted benchmarks, as can be inferred by inspection of Figure 8. These graphs demon-

[4]This ensures that there is no local state stored on the file system server, and removes the need to distribute the state for open files.

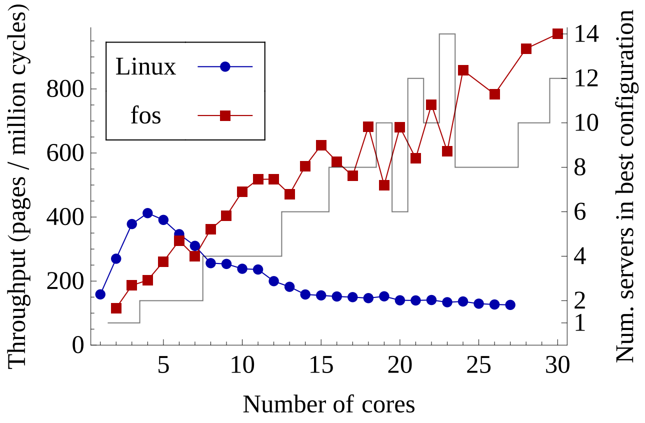


Figure 11: Comparison of fos and Linux for differently-sized multicores. System throughput is plotted against the number of cores consumed in the test (including clients and servers for fos). For fos, the number of servers at each point is chosen that yields best performance. The gray plot indicates the number of servers chosen in each configuration.

strate that fleets achieve much better performance and scalability than Linux services for large numbers of cores. Linux performs better for small numbers of cores, where time multiplexing the OS and application is a clear win. But in all of our microbenchmarks, Linux displayed inferior scalability when going beyond a few cores, and in two cases serious performance degradation. Here fos's design proved superior. Perhaps surprisingly, the transition where fos beats Linux occurs at a fairly small number of cores – smaller than currently available multicores.

Data for fos in Figure 11 was generated by selecting the optimal tradeoff between servers and clients. The number of servers used for each point is indicated by the square, gray plot. The data indicates a roughly 1:1 ratio of OS and application cores. If this seems unbelievably high, bear in mind that these microbenchmarks stress the OS service more than a regular application. In reality, one would expect a single microbenchmark client to correspond to several application cores.

fos is able to achieve scalability and performance through fleets. By separating the OS from the application, performance interference between the two is eliminated. More importantly, fleet design ensures the service is designed as an independent, parallel, self-aware entity from square one. This leads to scalable, adaptable services — crucial qualities for multicore success.

7. RELATED WORK

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

Traditional microkernels include Mach [5] and L4 [21]. 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 [15] 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 [10] OS shares the spatial awareness aspect of fos, but does not address parallelization within a system server and focuses on smaller configuration systems. 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. Also, in this work we show the scalability of our system servers which was not demonstrated in previous Barrelfish [8] work.

The structure of how fos can proxy messages between different machines is similar to how Mach [5] implemented network proxies with the Network Message Server. Also, Helios's [22] notion of satellite kernels is similar to how fos can have one server make a function call to a server on a different machine.

Disco [11] and Cellular Disco [17] run multiple cooperating virtual machines on a single multiprocessor system. fos's spatial distribution of fleet resources is similar to the way that different VM system services communicate within Cellular Disco. Disco and Cellular Disco argue leveraging traditional OSs 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 fixed boundaries imposed by VM boundaries can impair dynamic resource allocation.

fos bears much similarity to distributed OSs such as Amoeba [28], Sprite [23], and Clouds [13]. 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 OSs where much of the computation took place on student's desktop machines.

The manner in which fos parallelizes system services into fleets of cooperating servers is inspired by distributed Internet services. For instance, 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. In future work, we hope to leverage many distributed shared state techniques such as those in peer-to-peer and distributed hash tables such as Bit Torrent [12] and Chord [27]. fos also takes inspiration from distributed services such as distributed file systems such as AFS [25], OceanStore [20] and the Google File System [16].

fos differs from existing cloud computing solutions in several aspects. Cloud (*IaaS*) systems, such as Amazon's Elastic compute cloud (EC2) [1] and VMWare's VCloud, 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 [24] provide automatic cloud management and load balancing tools, but they are application-specific, whereas fos provides these features in an application agnostic manner.

8. CONCLUSION

New computer architectures are forcing software to be designed in a parallel fashion. Traditional monolithic OSs are not suitable for this design and become a hindrance to the application when the number of cores scales up. The unique design of fos allows the OS to scale to high core counts without interfering with the application. Furthermore, the abstractions provided allow an application written to our programming model to seamlessly span across a cloud just as it would across a multicore chip. We have demonstrated that this design is suitable for the multicore architectures of the future while providing better scalability than existing solutions. While a functional base system has been presented here, there still remains many areas open for exploration.

9. REFERENCES

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

[2] Tilera Announces the World's First 100-core Processor with the New TILE-Gx Family, Oct. 2009. http://www.tilera.com/.

[3] AMD Opteron 6000 Series Press Release, Mar. 2010. http://www.amd.com/us/press-releases/Pages/amd-sets-the-new-standard-29mar2010.aspx.

[4] GCC-XML project, 2010. http://www.gccxml.org/.

[5] 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.

[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] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[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] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175 – 198, May 1991.

[10] 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.

[11] 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.

[12] B. Cohen. Incentives build robustness in bittorrent, 2003.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March-April 2006.

[19] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, 7-11 2010.

[20] 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.

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

[22] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2009. ACM.

[23] 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.

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

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

[26] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, June 2010.