Message Passing in a Factored OS

by

Adam M. Belay

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2011
[June 2011]

Author ....
Department of Electrical Engineering and Computer Science
May 18, 2011

Certified by ....
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Message Passing in a Factored OS

by

Adam M. Belay

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The fos project aims to build a next generation operating system (OS) for clouds and manycores with hundreds or thousands of cores. The key design tenant of fos is to break operating system services into collections of distributed system processes that communicate with message passing rather than shared memory. This microkernel-based design has the potential for better performance and more efficient use of architectural resources. This is because each isolated address space can be scheduled on a separate dedicated core — It is anticipated that assigning entire cores to specific system processes will become feasible in the near future given the trend of increasing abundance of cores per die and in the datacenter. Cache locality in particular benefits from this approach, as user applications no longer compete with the operating system for on-core cache resources. However, for such benefits to be fully realized, the message passing system must be sufficiently low latency. Otherwise, too much time will be spent transfering requests between cores.

For this thesis, a high-performance message passing system for fos is developed and evaluated. The system supports a variety of messaging mechanisms, including a kernel messaging transport optimized for one-off communications, a low latency user messaging transport tailored to more frequent communications, and inter-machine messaging over TCP network sockets.

Experiments show that the user messaging transport can be used to make core-to-core system calls with comparable latency to the trap-and-enter procedure of conventional system calls in monolithic OSes. Thus the latency of messaging to a different core is sufficiently low to allow for the locality benefits of fos's distributed OS services to overshadow any negative impact of messaging costs.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Anant Agarwal for supervising this thesis and for helping me to become a better researcher. I am very grateful for the support and guidance he has provided, and I value his words of wisdom on how to present research effectively. I would also like to thank David Wentzlaff for his feedback on the research in this thesis and for his help in shaping the direction of my research in general. I am grateful for the time he spent, often late into the night, helping with the cache study, which is included as a chapter in this thesis.

This project would not have been possible without the efforts of the fos team and the Carbon Research Group, as they provided a solid foundation for me to develop my research. I am grateful for the many discussions on the messaging system design and for the team's willingness to put up with me while I was swamped at the end of the semester.

I would like to thank Patrick Winston for his sage advice throughout my academic journey. I also appreciate Anne Hunter's dedication to making sure I progressed through MIT smoothly. Finally, I would like to thank my parents for their support and for their confidence in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Factored OS (fos) is a new operating system (OS) for multicores and the cloud. Its construction is motivated by two challenges. First, as the number of cores per die continues to increase, it is becoming more difficult to maintain OS scalability. Second, the services that form today's cloud infrastructure tend to be fragmented because they each provide their own isolated interfaces.

The fos project addresses these challenges with the creation of a cloud-aware distributed OS that communicates with message passing. Through message passing, fos makes more efficient use of architectural resources and provides a uniform interface to all services. A unique aspect of fos is that it dedicates entire cores to each service. As a result, message passing does not incur the context switching overhead of traditional microkernels. Instead, data is passed efficiently between concurrent processes running on different cores.

This thesis presents the fos message passing system. Message communication is provided through mailboxes. A mailbox is a named endpoint that can be used to establish a one-way channel of communication with another process. In the event the intended mailbox resides on a different machine than that of the sender, a proxy service redirects messages over the network. This allows distributed OS services to scale beyond the confines of a single machine. In other words, a single messaging mechanism can be used both for multiple machines in a cloud data center and for multiple cores within a single machine. Thus, fos provides a more cohesive communication mechanism than the ad-hoc methods traditionally used in today's cloud services.

For local messaging on cache coherent multiprocessors, two separate transports have been implemented. The first is kernel messaging. Kernel messaging invokes the kernel to transfer message data to a heap data structure in the receiving process's address space. The second is user messaging. User messaging relies on a careful discipline over a shared memory channel and does not require entry into the kernel. This form of messaging takes more cycles to set up than fos's kernel messaging, but provides lower latency and better throughput. fos is also designed to be compatible with a fourth messaging transport in the near future: architectural support for message passing. It is anticipated that this mechanism will provide the best possible performance, and will obviate the need for kernel-traps to handle outgoing messages, similar to fos's current user messaging implementation.

fos schedules system services spatially rather than relying on exclusively temporal multiplexing. It dedicates entire cores (when available) to OS services. Such a scheduling policy has the potential to reduce the OS's competition for architectural resources with the user-level applications that run on top of it. In contrast, monolithic operating systems like Linux, invoke the OS on the same core as the application, either in response to system calls or to asynchronous interrupts. This causes the operating system to fight with applications for branch predictor state, TLB caches, and most significantly instruction and data memory caches.

A new tool, called CachEMU, was constructed to quantize the effects of OS pollution on memory caches. CachEMU is built on top of QEMU's binary translator and supports full-system instrumentation. Using data gathered with CachEMU, it is shown that there is potential for fos's design to better utilize on-die cache resources. In order for these benefits to be fully realized, however, fos's message passing system must have similar latency to traditional trap-and-enter system calls. To this end, the performance of fos's message passing system is fully evaluated, and it is shown to not be prohibitively expensive.

1.1 Contributions

The locality benefits of the fos design are characterized, and a message passing system that supports the fos design is presented. More specifically, the following contributions are made in this thesis:

- **Cache Performance Study.** CachEMU uses dynamic binary translation to provide real-time full-system memory trace output. The trace output is then piped to architectural models in order to simulate various cache configurations. These models show the benefits of running the OS and the application on separate cores. CachEMU has been made available as an open source tool.

- **Fast Messaging Implementation.** A highly optimized user messaging service, inspired by Barrelfish, is presented. A novel flow control mechanism allows the receiver to limit the rate that new messages arrive.

- **Dynamic Transport Selection.** Using competitive messaging, fos switches between kernel and user messaging dynamically in order to better optimize performance and memory usage.

- **Multi-machine Messaging** A single unified interface allows processes to send messages in a consistent way regardless of whether they are destined for a process running on the same machine or a remote machine. A proxy service aids in redirecting messages destined for remote machines.

- **Messaging Performance Analysis.** The performance of messaging is evaluated on cache coherent multicores, demonstrating that messaging can be extremely fast on contemporary hardware platforms, competitive with the local system calls used by monolithic operating systems. Moreover, a minimal, highly optimized IPC implementation is presented, establishing a lower bound on achievable messaging latency.

1.2 Outline

This thesis focuses on the design and performance of fos and its messaging system. Chapter 2 provides background information on the fos project, previous message passing OSes, and hardware support for messaging. Chapter 3 analyzes the cache behavior of Linux and shows the potential performance improvements that are possible through transitioning to a fos-like OS model. Chapter 4 provides an in-depth description of fos's messaging design and implementation. Chapter 5 characterizes the performance of message passing in fos. Finally, Chapter 6 concludes the thesis with a summary and a discussion of future directions.

Chapter 2

Factored OS and Message Passing

This chapter presents related work and background information on fos, fast message passing techniques, and hardware-based message passing.

2.1 fos: Design and Implementation

fos is a next generation OS for future multicores and today's cloud computing data centers [30]. It aims to address two emerging challenges.

First, the number of cores per die has been steadily increasing to unprecedented levels. For example, AMD recently unveiled a 16 core server processor as part of its "Bulldozer" architecture. In the embedded space, Tilera has developed a commercially available 64 core processor [5]. Furthermore, Intel has constructed a research prototype called the "single-chip cloud computer" (SCC) with 48 cores and software addressable on-chip memory [20]. One interesting aspect of the Intel architecture in particular is that hardware cache coherency is unavailable, necessitating that its systems software be constructed with message passing.

This trend of increasing core counts creates new challenges for the construction of OSes. The foremost concern is scalability. Although GNU/Linux has been shown to scale for up to 48 cores with some modifications [12], it is unclear if a monolithic kernel will be an adequate design for 100 or 1000 cores. Chapter 3 elaborates on this issue further, especially from the perspective of off-chip bandwidth usage and cache locality.

fos's solution to the scalability challenge is to construct an operating system with components that communicate with message passing — User applications are free to use either message passing or shared memory when it is available. Message based communication has the advantage of making data sharing explicit, rather than the implicit communication nature of shared memory. An additional advantage of message passing is that it is inherently compatible with traditional network protocols like TCP/IP. To this end, OS services are constructed using a fleet design pattern [31], or collections of distributed OS services that are highly scalable and adaptable. Several fleets have been implemented in the current fos prototype, including a naming service that provides information about the location of messaging mailboxes, a filesystem and networking service for I/O, and a page allocation and process management service for mediating access to memory and computation.

The second challenge is that system as a service (SaaS) cloud providers use virtualization to multiplex access to physical machines. This forces heterogeneity in the communication mechanisms used to build cloud applications. Specifically, processes within a virtual machine can use shared memory or pipes to communication while processes cooperating in separate virtual machines must rely on network sockets. Sockets can have needlessly high overhead, especially in the case where two virtual machines share the same physical machine. Some work has been done to mitigate these performance costs and to provide better interfaces [28,33]. However, more could be done to simplify cloud application development through providing a uniform communication mechanism for both local and remote communication.

fos addresses this issue by presenting a single messaging interface and namespace for all types of communication, similar to the uniform distributed namespace of Plan 9 [24]. fos goes further by providing a single system image; an application running under fos can scale beyond a single machine provided that it uses message passing to share data. Figure 2-1 illustrates a scenario where an application scales across several machines while communicating with a filesystem residing on a single machine. The application writer need not be concerned with the number of machines being used or the network locations of fos services. Furthermore, a library OS implementation called *libfos* hides the message passing details and provides a more traditional OS interface. The standard C library has been ported to run on top of libfos, and thus even legacy POSIX applications can be made compatible with fos through only minor modifications.

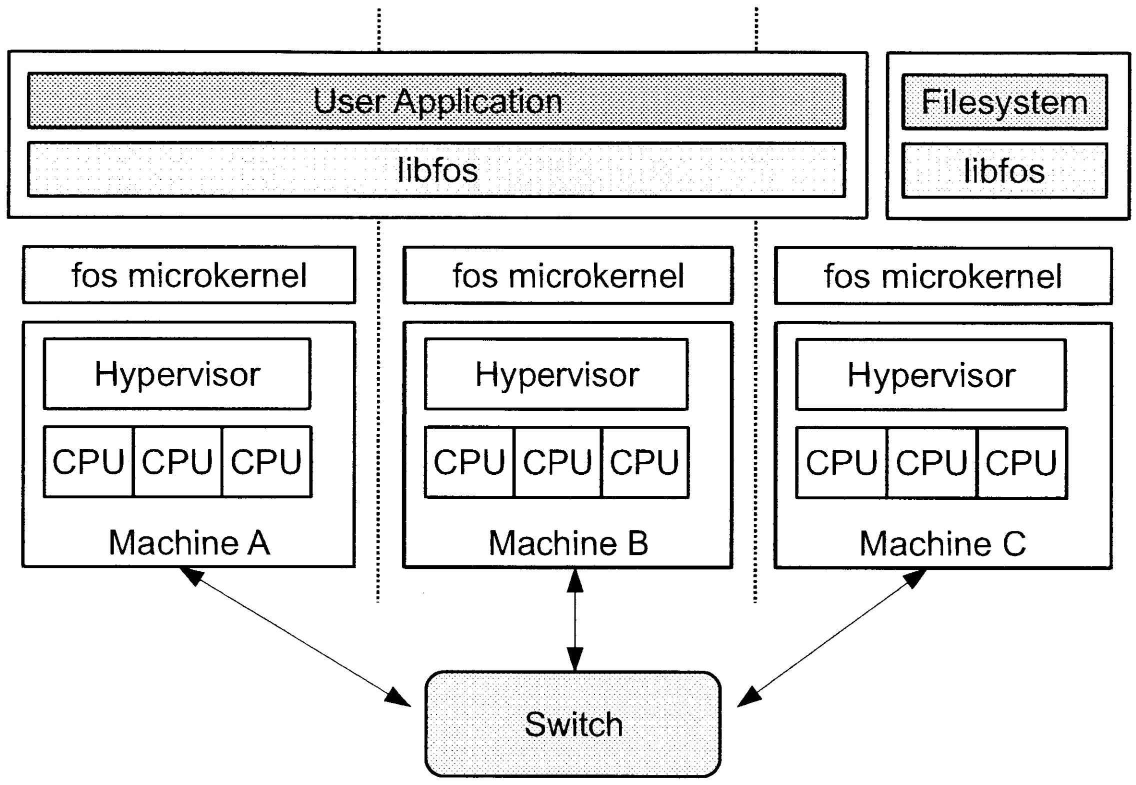

Figure 2-1: A view of fos running on three separate physical machines in a cloud datacenter. For this example, the application requires more cores than a single machine can provide. As a result, a single image, powered by libfos, maintains the illusion that the application is running on one large machine. The application relies on a filesystem service that is provided by the fos OS. Communication with the filesystem is seemless and uniform, regardless of the physical machine that initiated the request.

A limitation in fos's single system image is that memory cache coherency is not supported across machines. Software based cache-coherent shared memory is a current area of exploration and may be included in future releases of fos. This would enable threaded and more generally shared memory applications to gain the multi-machine scaling benefits that message passing applications currently enjoy. One potentially promising approach is multigrain shared memory [32], where fast fine grain cache-line sharing is used when available in hardware and course grain page-sized lines are sent across the network when misses occur between machine nodes.

2.1.1 System Architecture

The fos system architecture consists of a microkernel and a collection of distributed OS services that work together to provide more comprehensive OS functionality. Access to these components is abstracted by a library OS implementation called libfos. Figure 2-2 shows a high-level overview of fos running on a hypothetical multicore processor.

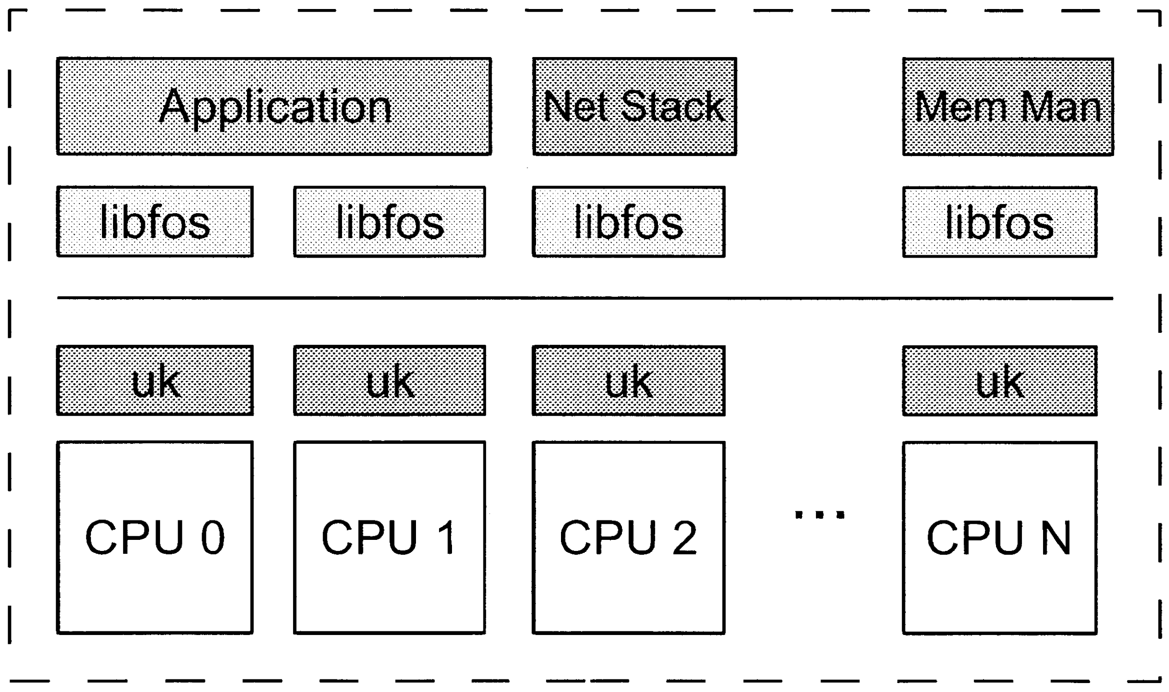

Figure 2-2: An overview of the fos system architecture. Each core runs a microkernel in privileged mode and libfos in unprivileged mode. Higher-level OS services run in separate processes and are accessed through message passing. Two example services are shown, a network stack and a memory manager.

fos's microkernel is minimal, providing only secure access to hardware, kernel messaging, shared memory setup for user messaging, and basic time multiplexing. All other functionality is implemented in userspace. Access to microkernel functionality is restricted through a capability system. Only processes possessing a secret nonce value for a given mailbox can enqueue messages.

The fos microkernel is implemented as a 64-bit x86 guest on top of Xen. This includes support for Xen's paravirt interface, allowing for more efficient virtualization of memory protection and processor control. Additional architecture ports are expected in the near future. However, it is anticipated that virtual machine platforms will continue to play a significant role in fos, as they are a natural unit of elastic computation for cloud service providers.

Libfos serves the critical role of simplifying access to fos's distributed services by providing convenient OS abstractions. For example, a file descriptor interface is provided by libfos, hiding the complexity of mailbox communication with the file systems server. Libfos also assists in optimizing system performance. For instance, libfos caches name service lookups and block device reads. Many significant components of the fos message passing system reside in libfos, including most of the user messaging transport.

2.1.2 Fleet Services

Most OS services provided by fos are implemented as fleets. A fleet is a set of distributed processes that work together to provide an OS service. In many ways, fleets are inspired by todays Internet services. In particular, the fos naming service maps mailbox aliases to mailbox locations much like DNS [21] maps hostnames to IP addresses.

The key design goal of fleets is to maximize scalability. This is achieved by using standard distributed system building techniques, such as replication and load balancing. Moreover, because fos fleets run on a dedicated set of cores, they engage in less data sharing and achieve better locality than if their respective service was running on all available cores, as is typical for monolithic kernels.

Another design goal of fleets is elasticity. To this end, each fleet service dynamically determines the number of cores it uses to process requests and perform work. One approach is to use Heartbeats [17] to monitor the response rate of a fleet. The system can then adjust the number of cores assigned to that fleet in order to maintain a desired quality of service.

The following fleet services have been implemented or are currently in development as part of the fos OS.

- **Naming:** The fos naming service provides a mapping between human-readable names called aliases and mailbox locations.
- **Filesystem:** The fos filesystem provides ext2 support. It is currently single-threaded. A more advanced filesystem is in active development.
- **Networking:** fos has a network fleet that is fully parallelized. It provides TCP/IP support as well as support for protocols such as DHCP and ARP.
- **Page Allocation:** The page allocation service manages access to physical memory pages. It is constructed on top of a highly scalable pool data structure.
- **Memory Management:** A dedicated fos fleet manages per-process virtual memory assignments. It is also responsible for servicing page faults.
- **Process Management:** The process management fleet includes a built-in ELF parser. It manages register state and assists in launching new processes.

2.2 Message Passing Optimizations

Message passing performance issues were a major drawback of microkernels when they were initially introduced (e.g. Mach's first implementation had large IPC overheads [1].) This is because each message incurs a context switching cost whenever the target OS service is running in a separate protection domain. Such costs discourage the creation of fine-grain protection domains, instead suggesting that OS functionality should be clumped together into larger pieces sharing a single address space. Taken to the extreme, a monolithic operating system with a single protection domain could be seen as an inherently more performant design. A variety of techniques have been explored in order to make microkernels with multiple protection domains as fast as single protection domain OSes. Principally, these efforts focused on making large reductions in message passing and context switching latency, assuming that the OS and the application would multiplex a single core.

More recently, with the advent of multicores, the cost of context switching between protection domains on a single core is no longer the central concern. Rather, the latency of cross-core communication is becoming a limiting factor. Fortunately, many of the optimizations that were developed, both for kernel-level and user-level message passing and context switching, are equally relevant toward this new challenge introduced in modern architectures.

This section provides an overview of specific techniques for efficient shared memory message passing, including optimizations for kernel-level messaging, user-level messaging, and large transfers between address spaces.

2.2.1 Kernel Messaging Optimizations

Traditionally messaging is provided as a kernel-level service. L4 was one of the first operating systems that offered fast message passing performance through the kernel [18]. This was accomplished through a variety of optimization techniques. First, L4 stores small messages directly in processor registers, avoiding the need to read such messages from memory. This optimization is not possible when messaging between different cores, because registers cannot be shared between cores directly. Second, the L4 microkernel sets up temporary virtual memory mappings (exposed only to the kernel) that allow it to copy the message payload directly into the receiver's address space. Great care must be taken to ensure the current TLB reflects the mappings accurately and that TLB updates are not prohibitively expensive. Third, the instruction stream is optimized to take conditional jumps faster in the case of a valid messaging request. Fourth, a flat memory model is used in order to avoid the need to save and restore segment registers. L4 provides a fully synchronous message passing paradigm and is optimized for the case of where the process context on a single core needs to be changed in order to service the message. In contrast, fos requires asynchronous messaging and will typically send core-to-core messages that require no context changes. Nevertheless, many of the optimizations in L4 could be relevant to improving fos's messaging performance simply because they are effective at reducing the cost of system call entry and exit.

An earlier approach to fast kernel-level messaging, targetting the DEC Firefly architecture, is LRPC [7]. In many ways it inspired the implementation of L4. One interesting optimization is that the application and the OS service can share a stack during a system call.

2.2.2 User Messaging Optimizations

User-level message passing is an alternate approach where the kernel only performs context switching (for cases that require multiplexing a single core) and memory setup. All other aspects of message passing are performed at user-level. This notion was pioneered in URPC [8]. URPC combines an efficient user-level message passing scheme with user-level thread scheduling. Messages are sent and received by the application directly over a shared memory mapping established before the first message is sent.

Barrelfish [3] is constructed using both LRPC and URPC messaging mechanisms. Its URPC implemention is particularly efficient because it uses a scheme that invalidates only a single cache line for sufficiently small messages. This is achieved by sending data in cache-line sized chunks and reserving the last few bytes for a generation number. The receiver polls the next cache line it expects to be available, waiting for the generation number to be incremented to the next number in the sequence. Since the generation number is updated last, this ensures that all of the remaining data in the line is available. Thus the receiver fetches both data and book keeping in a single round of the cache coherency protocol. This optimization is leveraged in the fos user-level messaging implementation.

2.2.3 Memory Remapping

For very large data transfers between protection domains, the overhead of copying bytes, even across a shared memory mapping, can be high. An alternative approach is to use fast buffers (fbufs) [15]. In fbufs, memory remapping is used to securely transfer ownership of entire pages of data. Specifically, the sender writes data into pages of memory and then relinquishes ownership of those pages to the receiver. This is beneficial not only because the sender can achieve zero-copy transfers, but also because fbuf operations tend to pollute the sending core's data cache less.

2.3 Hardware Message Passing

Some emerging multi-core architectures are providing light-weight hardware support for message passing. For example, Tilera's Tile64 allows unprivileged code to pass messages directly through the interconnect network via specialized processor registers [29]. Moreover it can deliver interrupts to notify userspace processes of message arrival. This allows it to support both polling and interrupt-based delivery. Tilera's messaging hardware completely bypasses shared memory access and does not rely on cache coherency protocols. As a result, it has the potential to perform better than even highly optimized user and kernel based message passing methods.

Intel's SCC also provides an interesting hardware-assisted message passing mechanism [20]. Efficient transfer of data between cores is enabled by a region of ultra-fast on-chip memory called the message passing transfer buffer (MPTB). A MPTB is present on each tile along with a set of test-and-set registers. The test-and-set registers are used to implement fast synchronization primitives, a necessary mechanism for supporting atomicity. Coherence is managed entirely by the programmer and without explicit hardware assistance. A communication library called RCCE provides a more natural programming model by supporting put and get operations. These operations are fast, as they copy data directly from the sending core's L1 cache to the receiving core's MPTB.

Chapter 3

Cache Performance Study

An emerging question in OS research is whether it will be necessary to adopt new designs in order to support the next generation of multicores. This especially pressing considering that processors could have hundreds or thousands of cores in the near future. Most contemporary OSes, including the popular Linux and Windows OSes have adopted a monolithic design. In this case, each application shares its core with the OS kernel. Specifically, the kernel steals cycles from the application during interrupts and in response to system calls. Such a design has the potential to introduce greater competition over architectural resources. An alternative design, as adopted by fos, is to dedicate entire cores to the OS. This allows the OS and application to run separately while still permitting interaction through message passing.

This chapter presents a study of cache performance, comparing the memory behavior of the different OS designs. A new open source memory trace generator and cache simulator called CachEMU was created to evaluate the inherent advantages and disadvantages of each design in terms of memory access costs. It is crucial for future operating systems to use memory resources efficiently in order to avoid higher latencies and to conserve off-chip bandwidth [13]. Otherwise, poor cache hit rates could cause cores to loose excessive cycles waiting for memory references. The results in this study indicate that a fos-style OS accesses memory more efficiently through improved cache locality. Thus, it has the potential to be a superior design for future multicores.

3.1 Overview

There are two competing factors that determine the cache performance differences in each OS design. The first is *competition* and favors running the OS and the application on separate cores. The second is *cooperation* and favors running the OS and application on the same core.

Competition is an effect where the OS and the application fight over the architectural resources of a core, including memory caches, TLBs, branch predictor state, registers, and other stateful mechanisms. This effect results in performance losses because lost architectural state must be recalculated either as a dependency for computation to proceed or as necessary background information to perform successful speculative execution. The results gathered with CachEMU suggest that fos and other scalable OSes [3, 10] have inherent performance advantages because their design results in a reduction in *competition*. Specifically, because each core is dedicated to a specific task (belonging to either the application or the OS,) the working set size on any given core is reduced to better fit within the capacity of the local cache as well as other architectural resources.

On the other hand, running the OS and application on separate cores causes an increase in overhead associated with *cooperation*. *Cooperation* occurs whenever the OS and application share memory. For example, during a system call, the application might pass a cache hot buffer to the kernel as an argument. If the OS were running on a different core, the shared state would have to be transfer either using the cache coherency protocol or with hardware message passing. On the same core, however, the shared state could remain in the core's local cache. Though certainly impactful, the added *cooperation* costs typically do not exceed the benefits of reducing *competition* by running the OS on its own core. The reason is that the OS can be placed on a core that is near the application core, resulting in the potential for more efficient data transfers. For example, a shared L3 cache between cores could prevent the need to go off-chip to retrieve shared data.

3.2 Results

Using CachEMU, a variety of experiments that examine the effects of competition and cooperation in the context of OS and application workload placement strategies are presented.

3.2.1 CachEMU

CachEMU is a memory reference trace generator and cache simulator based on the QEMU [6] processor emulator. Through modifications to QEMU's dynamic binary translator, CachEMU interposes on data and instruction memory access. This is achieved by injecting additional micro-operations at the beginning of each guest instruction and data lookup. CachEMU's cache model has fully configurable cache size, associativity, block size, and type (instruction, data, or unified). Raw memory tracing, however, could easily be directed to additional purposes in the future, such as write-back buffer modeling or simulating TLB costs. Like SimOS [25], CachEMU's modeling can be enabled or disabled dynamically, allowing system boot-up or any other operations that are not relevant to an experiment to run without added overhead.

Several of QEMU's advantages are preserved in CachEMU, including the ability to perform full-system emulation across multiple processor ISA's with realistic hardware interfaces. Although this study is currently limited to Linux on x86-64, full-system emulation makes CachEMU a powerful tool for studying the effects of OS interface on a variety of potential platforms (e.g. Android mobile phones and Windows desktops.) Simics provides similar full-system and memory trace capabilities to CachEMU but is currently proprietary [19].

CachEMU builds upon the work of past memory reference tracers. For example, ATUM used modifications to processor microcode to record memory traces to a part of main memory [2]. Previous studies have established that OS interference can have a significant effect on memory caches [2, 14].

CachEMU brings the ability to study OS interference to new machines and allows for the study of new kinds of applications. Such an effort is relevant now more than ever because of the dramatic increase in cores on a chip, each with dedicated cache resources, changes in the scale of applications, and the rise in complexity of cache topologies.

3.2.2 Methodology

CachEMU was used to evaluate the effects of competition and cooperation, counting kernel instructions toward the OS and user instructions toward the application. Each application was given its own virtual machine with a 64-bit version of Debian Lenny installed and a single virtual CPU. Instruction-based timekeeping, where each guest instruction is counted as a clock tick, was used with QEMU in order to mask the disparity between host time and virtual time. For cases where the user and the OS were using separate caches, a basic cache coherency protocol simulated shared memory contention by having writes in one cache trigger evictions in the other cache. Evictions were performed only for unshared caches. For example, a memory reference could cause an eviction in a separate L2 cache while leaving the entry present in a shared L3 cache. All cache accesses were modeled with physical addresses and each cache line used a standard 64 byte block size.

Five common Linux workloads were chosen because of their heavy usage of OS services. They are as follows:

- **Apache:** The Apache Web Server, running an Apache Bench test over localhost.
- **Find:** The Unix search tool, walking the entire filesystem.
- **Make:** The Unix build tool, compiling the standard library 'fontconfig' (includes gcc invocations and other scripts.)
- **Psearchy:** A parallel search indexer included with Mosbench [11], indexing the entire Linux Kernel source tree.

- **Zip:** The standard compressed archive tool, packing the entire Linux Kernel source tree into a zip archive.

3.2.3 Cache Behavior

In order to gain a better understanding of the effects of capacity on competition and cooperation, a spectrum of single-level 8-way associative cache sizes were tested ranging from 4KB to 16MB. For each test, the number of misses occurring under separate OS and application caches was compared with the number of misses occurring in a shared application and OS cache. In general, it was observed that competition was a dominant factor that discouraged sharing for small cache sizes, while cooperation was a dominant factor that encouraged sharing for larger cache sizes.

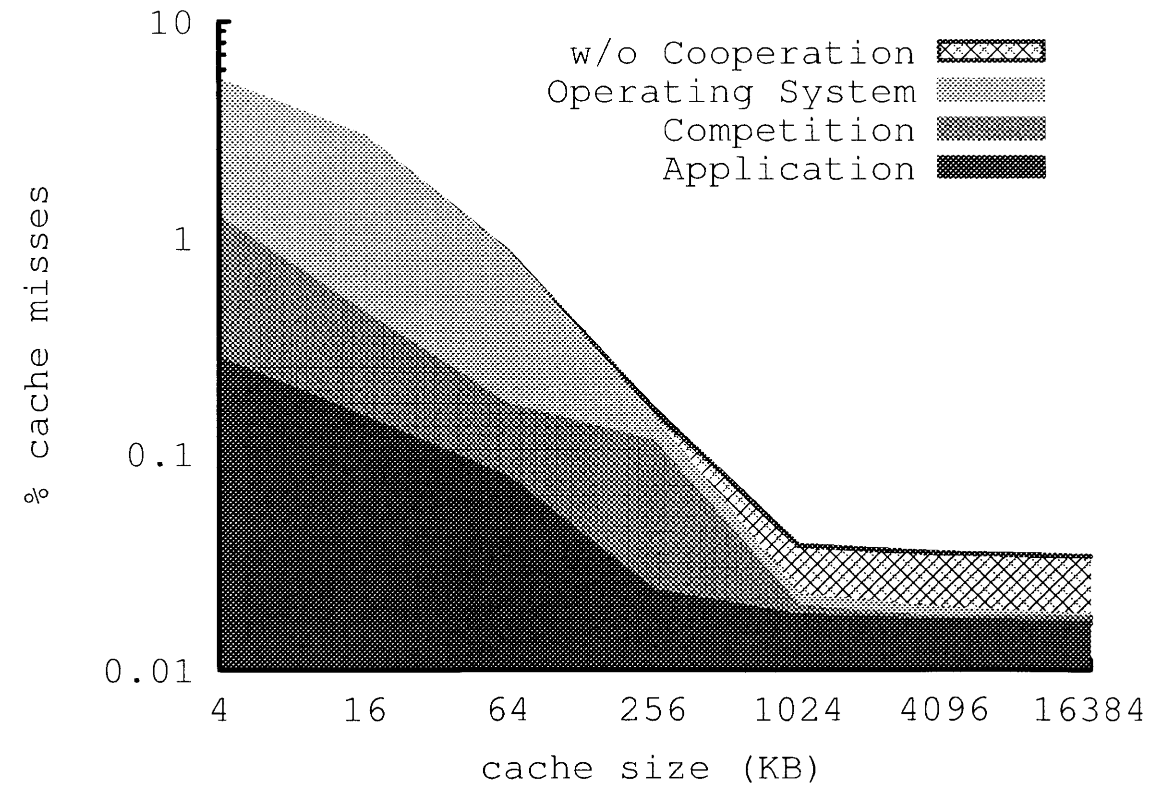

Figure 3-1: Cache Miss Rate vs Cache Size for the zip application. Shows shared cache misses attributable to the OS and Application alone as well as the Competition between them. Also shows additional misses that would occur without the Cooperation benefits of a shared cache.

For example, figure 3-1 shows the cache behavior of the zip workload when the OS and the application share a cache. For this test, competition effects were dominant until the cache size reached 1 MB. Then from 1 MB to 16 MB the reduction in misses because of cooperation — shared data between the application and the OS — overtook the number of cache misses caused by competition. Although the number of misses avoided as a result of cooperation is relatively small, the performance impact is still great because for larger cache and memory sizes (i.e. where cooperation is a dominant effect) there tends to be much greater access latencies. It is worth noting that the zip workload generally had a higher proportion of misses caused by the OS, a common trend observed in all of the test workloads.

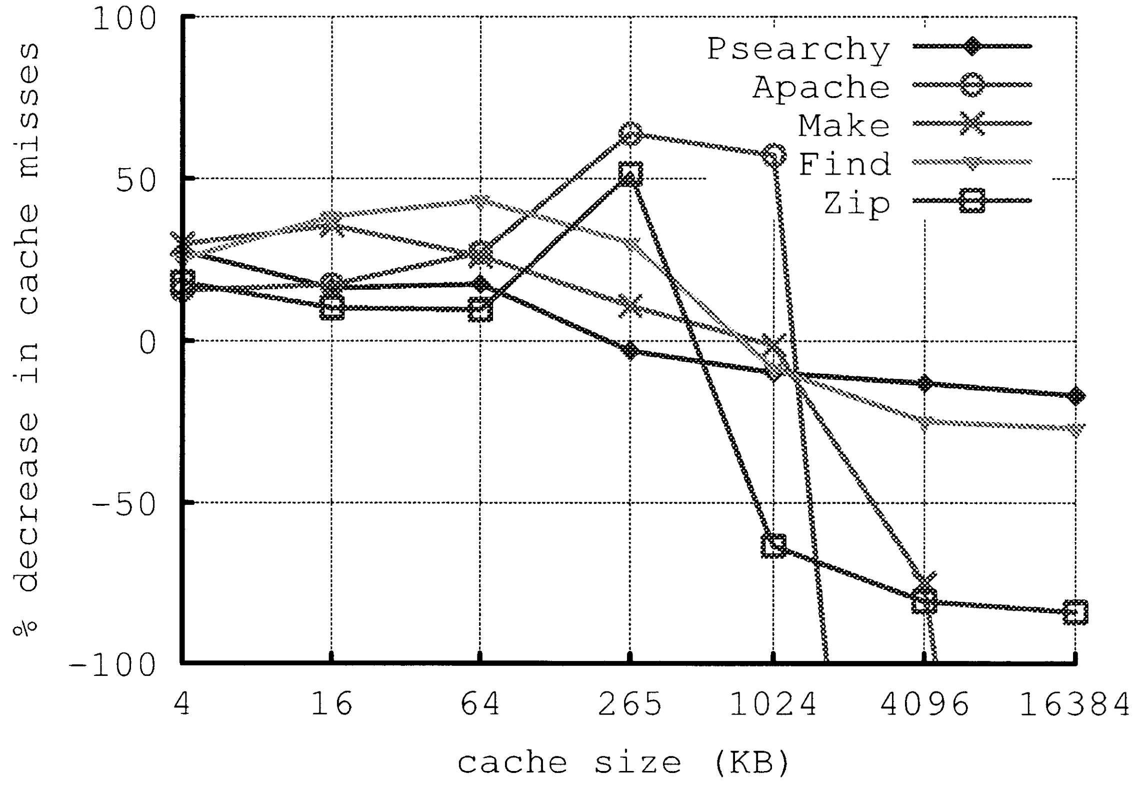

Figure 3-2: The percentage decrease in misses caused by splitting the application and OS into separate caches.

Figure 3-2 includes all five test applications and shows the effect of cache size from a different perspective; The percentage decrease in total misses caused by having separate caches was calculated. This normalizes the cache effects to the baseline miss rate of each cache size. For small cache sizes (usually less than 256 KB,) there were advantages to having separate OS and application caches because of the reduction in cache competition. For large cache sizes (1 MB and above,) data transfers between the OS and application became a dominant factor, and a net advantage was observed for having a shared OS and application cache. The behavior of caches between 256 KB and 1 MB was application specific and depended on working set size.

3.2.4 Performance Impact

The performance impact of OS placement on contemporary processors was studied by building a three-level Intel Nehalem cache model for cores clocked at 2.9 GHZ. The model includes separate L1 data (8-way associative) and instruction (4-way associative) caches, each with 32 KB capacity. For L2 and L3 a 256 KB 8-way associative cache and a 8 MB 16-way associative cache were modelled respectively. Cache latencies of 10 cycles for L2 access, 38 cycles for L3 access, and 191 cycles for local memory access [22] were assumed. Using these parameters, Figure 3-3 shows the additional cycles-per-instruction (CPI) due to the memory system for the following OS placements: all three cache layers shared, separate L1 and L2 but shared L3, and all three cache layers separate. Contention misses caused by shared state were modeled by adding the equivalent latency of the next higher shared cache level. This included using local memory latency when all higher cache levels were unshared. In practice, communication between processors on different dies can be slightly more expensive than local memory accesses [22], but this should nonetheless be a reasonable approximation.

CPI calculations were then used to estimate overall application speed up. Since actual non-memory CPI is workload dependent, and cannot be estimated by our simulator, a conservative value of 1.0, the median total CPI for the Pentium Pro [9], is assumed. Figure 3-4 shows projected performance improvements for each of the five workloads. Running the OS on a different core with a shared L3 cache was always better than running the OS on the same core, except for the Psearchy workload where it was equivalent.

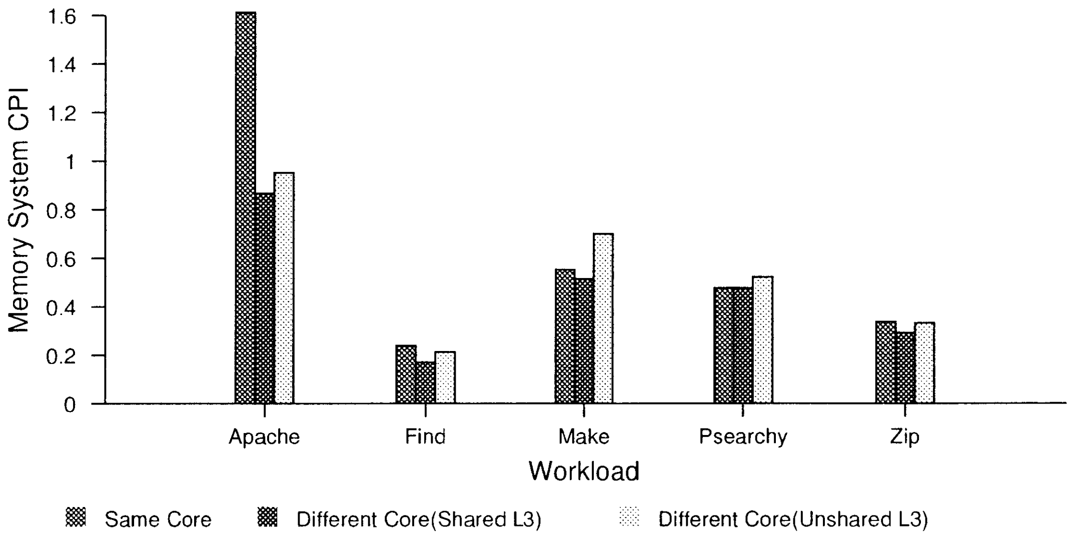

Figure 3-3: Memory related Clocks per Instruction (CPI) when executing OS and Application on same core, different cores but same chip, and different cores and different chip. This utilizes a model of the Nehalem cache architecture.

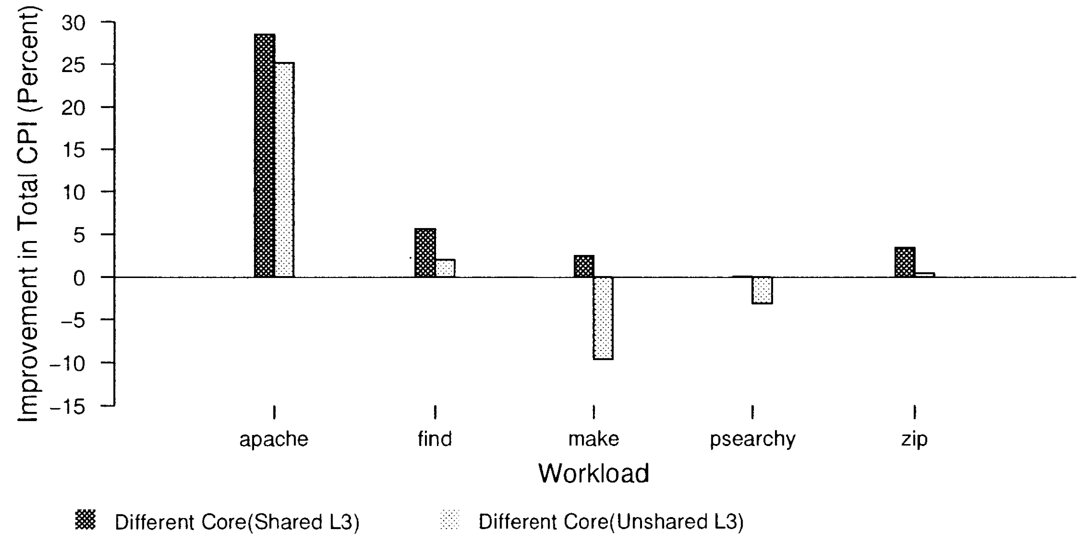

Figure 3-4: Overall percentage improvement in the CPI due to memory effects when the OS is run on a different core. The non-memory CPI of each workload is not known, so a conservative value of 1.0 is assumed.

3.3 Discussion

For five Linux test workloads, running the OS on a different core was always more efficient as long as the two cores shared an L3 cache. Thus, with proper scheduling, it appears that fos has the potential to better utilize multicore architectures than monolithic OS designs. Although branch predictor, TLB state, and registers were not modeled in CachEMU, including them would only further increase the competition costs of running the application and OS on the same core. Consequently, even better aggregate results for a fos placement strategy are possible.

FlexSC [26] also confirms this notation by showing that a modified version of Linux can gain these benefits as well by adopting the fos approach of placing the OS on a different core. However, CachEMU shows that cooperation and contention alone are compelling reasons to run the OS on a separate core, whereas FlexSC includes the additional effect of running the OS and application in parallel (using a custom thread scheduler) in its end-to-end performance analysis. Thus, the results obtained with CachEMU provide a more detailed study of the specific factors that contribute to performance advantages for fos's OS placement policy. Nonetheless, the ability to exploit parallelism between the OS and application, especially when the application can issue system calls asynchronously, is an additional performance advantage of running the OS and application on separate cores.

One drawback of dedicating an entire core to an OS service is that the core cannot be used for other purposes, whereas in a typical OS design all cores executes both application and OS instructions. Fortunately, since a core performing a particular OS task can potentially service multiple applications at once, this cost tends to be amortized. Moreover, future multicores may provide a number of cores that rivals the number of active processes in the system, leaving plenty of space and time to dedicate entire cores to OS functions.

An aspect that was not explored in this cache study is the cost of making a core-to-core system call. If messaging costs are comparable to the privilege change overhead of a local system call then placing the OS on a different core will have no extra costs beyond the cache effects described in this study. In Chapter 5 we show that this is in fact the case for nearby cores on contemporary architectures. Of course, hardware accelerated message passing could mitigate both performance concerns due to cooperation as well as the system call latency of messaging a separate core.

Chapter 4

Design and Implementation of Messaging

This chapter provides an overview of the fos messaging system design. It includes a description of user messaging, kernel messaging, and TCP/IP multi-machine messaging. Furthermore, the chapter presents a dynamic transport selection mechanism based on competitive messaging that chooses between using kernel or user messaging in order to optimize performance and memory usage.

4.1 Overview of Messaging in fos

In fos, messaging is a fundamental OS service available to every process. Processes receive messages through mailboxes. A mailbox is similar in concept to Mach's port abstraction [1] and essentially serves as a named messaging endpoint. Mailboxes are created and destroyed through dedicated system calls provided by the fos microkernel. However, a newly created mailbox must also be registered with the naming service [4] for discovery purposes.

The naming service essentially serves as a distributed hash table linking aliases, unique 128-bit key values, to mailbox locations. In order to register a mailbox with the name service, an alias must first be determined. Typically, these values will be calculated by hashing human readable strings. For example, the fos block service uses the string "`/sys/block_device_server/input`." The name service also supports indirect aliases, where one alias refers to a second alias that directly corresponds to a mailbox. This effectively allows multiple aliases to correspond to a single mailbox.

4.1.1 Messaging System Properties

A variety of decisions and tradeoffs can be made in a message passing system. fos's message passing system adheres to the following properties:

- All communication is connectionless. When stateful connections are required, they can be implemented at the application level.
- Several processes can enqueue messages to a single mailbox, but messages can only be dequeued by the specific process that registered the mailbox. This allows for both one-to-one and many-to-one communication patterns.
- Each mailbox has a fixed size buffer allocated to it. If the buffer is full, new messages are rejected until enough space becomes available.
- The receiver is guaranteed to receive messages in the same order that the sender enqueues them. However, the order of messages across multiple senders is unspecified.
- All messaging communication is asynchronous. In other words, a process can enqueue a new message before a previous message is dequeued.

A connectionless, many-to-one design was chosen over traditional channel based messaging (e.g. TCP/IP) for two reasons. First, it is expected to map more closely to hardware interfaces that support direct access to on-chip-network resources, resulting in a more raw and high-performance interface. Second, the reduction in messaging state could make it easier to support live process migration features in the future.

4.1.2 Security and Isolation

fos provides secure access to mailboxes through a capability system. Each mailbox has a secret 64-bit capability value associated with it. Only processes that supply the correct capability value can enqueue messages on a mailbox. This is enforced by a security check each time a message is sent.

Typically, a process sends capabilities and aliases over message passing in order to grant access to another process. A capability depot service provides any interested party with access to a set of public mailboxes provided by core fos fleets. This helps to resolve any bootstrapping issues.

Even though shared memory is used in some cases (i.e. user messaging,) isolation is fully preserved through careful access disciplines. More details on the security of user messaging are described in section 4.2.2.

4.2 Messaging Architecture and Implementation

A complete message passing system was implemented for fos. It consists of major components in libfos and in the microkernel as a set of system calls. Additionally, two fleets serve in a support capacity for fos messaging. The first is a distributed naming service [4] that provides a mapping between aliases and physical locations. A physical location is a pair consisting of a host address and a local identifier. The physical location can be used to determine whether the message destination is a process on the local machine or a process on a remote machine. In the event that a message is bound for a remote machine, the second fleet, the proxy service, is used to redirect traffic over the network using TCP/IP. A corresponding proxy service, running on the remote system, then passes the message to the target process using a local communication mechanism.

fos supports two local communication mechanisms, kernel messaging and user messaging. Kernel messaging uses a system call to copy message data across address spaces. In contrast, user messaging bypasses the kernel (in the steady-state) and uses a shared memory channel to communicate between processes. Figure 4-1 shows a

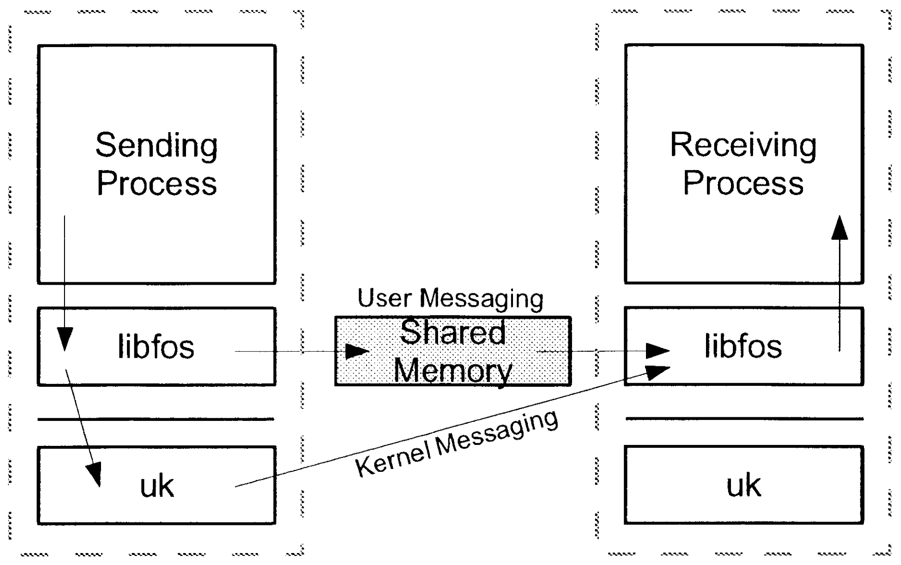

Figure 4-1: The paths that the two local messaging methods take through fos's software stack. User messaging passes data through a shared memory channel. Kernel messaging uses the microkernel to copy data across address spaces. A line demarcates processor privilege levels. For both mechanisms, the receiving process does not need to enter the kernel, and instead relies on libfos to detect and process new messages.

view of the path these two mechanisms take through the fos architecture.

fos's messaging support is implemented to be transport agnostic — the details of each transport are abstracted behind the fos messaging layer. This is beneficial because it allows for local messaging transports to be changed dynamically and transparently without disrupting or interfering with the application. Also, it allows fos to hide the distinction between local and remote messaging (e.g. over TCP/IP). Coupled with a single-system-image, this enables fos to present nearby and distant mailboxes as if they were each part of a single large machine.

4.2.1 Kernel Messaging

Kernel messaging uses the microkernel to transfer message data between processes. Thus, the microkernel is trusted to modify the receiver's address space without violating isolation.

Setup of kernel messaging occurs in libfos and subsequently the microkernel as follows. During mailbox creation, the receiving process allocates virtually contiguous memory off its local heap. This serves as a buffer for incoming messages. Next, libfos registers a pointer to the mailbox buffer with the microkernel. The microkernel then validates the address and stores it in a hash table indexed by the local identifier. The hash-table lies in shared kernel memory and is protected by a lock. If necessary, a more scalable lock-less design could be used in the future [16].

For sending messages, libfos makes a system call into the microkernel, providing a message buffer, length, and local identifier. The microkernel looks up the mailbox structure in the hash table and copies the message buffer to the address space of the mailbox's parent process. New messages are stored in the incoming message buffer using a specialized heap data structure. Because the incoming message buffer can be accessed by many sending processes in parallel, atomicity is ensured with a per-mailbox lock. The lock can be accessed by both user-space and kernel-space, so great care must be taken to prevent untrusted code from causing the microkernel to deadlock. This denial of service hazard is avoided by returning a try-again code whenever the lock is already taken, rather than blocking in the kernel and waiting for the lock to become available.

Kernel messaging has only a constant initial setup cost in time and memory per mailbox (i.e. registering the mailbox with the kernel). As a result, its overhead can be easily amortized, especially when several different processes are sending messages to the same mailbox. On the other hand, its per-message overhead is higher than user messaging because each message send operation must incur the overhead of trapping into the kernel.

4.2.2 User Messaging

User messaging relies on single-reader, single-writer, shared memory channel to transfer data between processes. Channels are created on-demand by libfos in response to message send requests. A specialized system call is used to request a channel connection to another process. It allocates memory and modifies the sender's page table to include a mapping of the channel. It also modifies a user-level memory region in the address space of the receiver, called the doorbell. The doorbell is an asynchronous upcall that serves to notify the receiver that a channel request is pending. When the receiver detects that the doorbell has been set, it invokes the accept channel system call. This system call completes the setup by mapping the channel in the receiver's address space. In the case of multiple pending requests, the accept channel system call is called repeatedly until no further pending channels remain. Finally, the doorbell bit is cleared.

Messages are encoded and decoded in a way that fully exploits the performance potential of hardware implementations of cache-coherent memory. Specifically, messages are transferred in increments of cache-line sized packets. Each packet consists of a data payload and a tag containing a generation number, acknowledgement number, and flags. The tag is placed at the end of the cache-line and is always written after the data payload. This allows the receiver to poll for the next packet by detecting a change in the generation number value. When the generation number matches the anticipated next value, the receiver knows that the packet is ready and that the full contents of the data payload are available. In most cases, only a single cache-line is invalidated per messaging packet, although in rare cases, the receiver may invalidate the cache-line before the the tag has been updated with the next generation number. For this event, the generation number check would prevent the receiver from proceeding until the sender has finished writing data to the packet. This method of packing messaging packets into a single cache-line is heavily inspired by a similar implementation in Barrelfish [3]. Although the shared memory channel is finite in size, it is treated as a circular ring buffer. Because a message may exceed the size of an individual packet, each message contains a special header (included in the first packet of the message) that informs libfos of the overall size of the message. Potentially unlimited size messages are supported as long as the receiver is actively dequeuing packets. Figure 4-2 shows the address layout of the packets in the shared memory channel.

For security reasons, the receiver allocates a buffer to store the complete contents of each message it dequeues from the shared memory channel. This prevents the message contents from being inappropriately modified by the sender, perhaps after

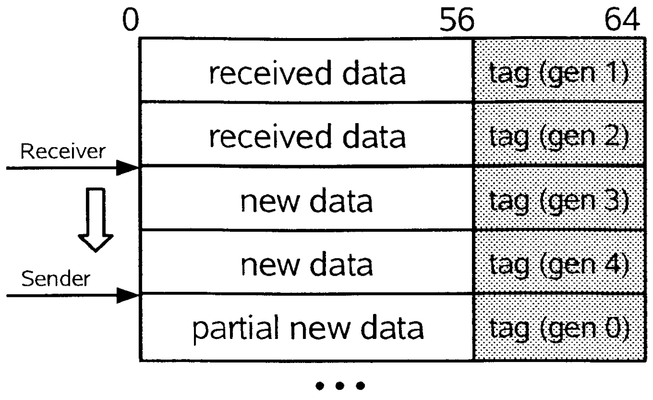

Figure 4-2: The address layout of the shared memory channel for 64-byte cache-lines (typical for x86 architectures). The arrows indicate position pointers in the ring buffer. In this example, the sender has written two packets and the receiver has processed two packets. A third packet is only partially written by the sender, and hence, has a stale generation number.

validation routines have been run by the receiving process. Moreover, packet memory can be freed once the message is copied to the buffer, making space immediately available for upcoming messages.

One innovative aspect of fos's user messaging is the way it implements flow control. Although fos messaging is unidirectional, flow control is nonetheless supported by having the receiver perform writes to the shared memory region. Specifically it modifies a dedicated acknowledgement byte reserved as part of the tag data in each packet. The receiver keeps a count of the number of packets it has dequeued. Whenever the count crosses a threshold relative to total packet capacity of the channel, the most recently received packet's acknowledgement section is set to the received count and the received count is cleared to zero. In a similar fashion, the sender keeps an outstanding packets count and increments it whenever it sends out a packet. Before using a new region of packet memory, the sender checks if there is a non-zero acknowledgement value in the tag. If so, it subtracts the acknowledgement number from its outstanding packet count. If ever the outstanding packet count is equal to the packet capacity, libfos will return a busy error when a sender attempts to enqueue a packet. Altogether, this mechanism prevents the sender from overwriting a packet slot before the receiver has had a chance to receive the old packet data.

Just as messaging channels are created dynamically, they are also destroyed dynamically. Channels are evicted by the receiver whenever they become infrequently used. This is critical because the receiver must poll all channels to determine if a new message is available. Keeping the number of channels limited helps to ensure that the polling latency does not grow larger over the lifetime of the receiving process.

4.2.3 Dynamic Transport Selection

Kernel and user messaging have different tradeoffs in performance and memory usage. For infrequent communication, kernel messaging is preferred because it requires less memory and does not incur the overhead of creating a shared memory channel. Setting up shared memory can be expensive because it requires modifications to the page tables on both cores. On the other hand, kernel messaging has greater latency than user messaging because it must incur the trap and enter overhead of a system call. Thus, for a longer period of sustained communication the better latency of user messaging makes up for its initial setup costs.

In fos, kernel and user messaging are switched between dynamically through a transport selection system using a competitive messaging algorithm. Under the assumption that communication will be infrequent, the first messages sent to a mailbox always use kernel messaging. A counter measures the number of messages destined for each mailbox. When the number crosses a threshold, the transport selector switches over to user messaging by creating a shared memory channel with the destination process. fos uses the latency of channel creation to set the threshold, such that the time spent sending the initial kernel messages roughly matches channel creation latency.

When a shared memory channel is idle for a long period of time, it can get evicted in response to the creation of a new channel. This removal of channels allows fos to switch back to kernel messaging automatically when it is better suited.

fos's ability to dynamically change messaging transports allows it to adjust to changing application communication patterns. This has the advantage of conserving resources and avoiding setup costs for rare communication paths, while achieving maximum performance for frequent communication paths.

4.2.4 Multi-machine Messaging

fos provides a single namespace that can span multiple cache-coherency domains or physical machines. All mailboxes in the namespace are accessible — given the proper credentials — to each member system, effectively creating a universal way of referencing resources across machines.

Figure 4-3 shows fos's messaging architecture for multi-machine support. A proxy fleet provides the service of forwarding messages across the network. Each proxy fleet has two types of member processes. The first is a proxy sender, which focuses on taking local messages and sending them over the network wire. The second is a proxy receiver, which receives requests and authenticates them before forwarding a message to a local process. These two component types work together to provide messaging across machines.

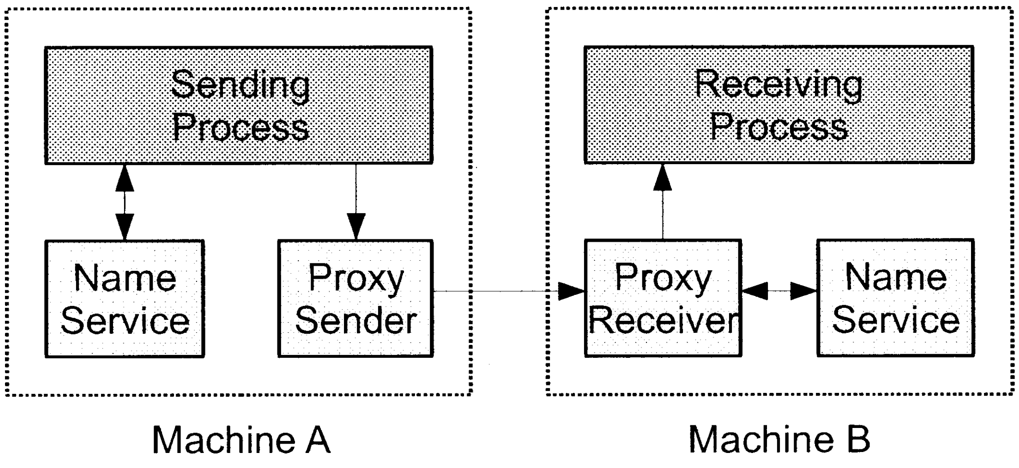

Figure 4-3: An overview of fos's multi-machine messaging infrastructure. When libfos (not shown) discovers that a message is bound for a remote machine through performing a name lookup with the naming service, it sends the proxy sender a remote message request. The sender then forwards the message over the network to a proxy receiver on the remote machine. The proxy receiver performs a second name lookup to verify the destination, and then it sends the message to the receiving process.

The networking service and libfos also play a significant role in multi-machine messaging. Whenever a message is sent to a new mailbox, a name service lookup occurs, returning a location. If the location does not correspond to the local machine, libfos forwards the messaging request to a proxy sender. Similarly, the proxy receiver uses its local naming service to validate the integrity of a message. Each request includes the final destination alias, and the name server is used to determine if such an alias was indeed exported by the local machine.

Like other fleets, the proxy service can have multiple processes working together, both as proxy senders and proxy receivers. In order to balance load, each libfos instance can use a different proxy sender. This form of elasticity is especially important in the case of multiple network interfaces, where a single core may not be able to saturate a link.

Chapter 5

Evaluation

The performance of the message passing system has a significant impact on fos's overall speed. Each application and fleet service relies on messaging to communicate with other processes. Thus, messaging is on the hot path of most operations performed in fos. In order to improve end-to-end performance, great effort was spent toward minimizing messaging latency. This chapter evaluates the performance of fos messaging. It also demonstrates the operation of dynamic transport switching by showing how fos can switch from kernel to user messaging in response to messaging activity. Finally, for the purpose of better understanding the potential for optimizing fos messaging further, performance is compared to a minimal highly optimized interprocess communication (IPC) implementation that approaches ideal performance.

5.1 Testing Procedure

Two different processor models, running in 64-bit mode, were used in the messaging performance experiments. The first processor is a four core Intel Xeon E5530. The second processor is a twelve core AMD Opteron 6168. Measurements were taken using the time stamp counter (TSC.) Because fos runs in a virtual machine, great care was taken to ensure that the TSC was not emulated by software but rather was provided directly by hardware. Another possible concern with virtualization is that the host kernel will interrupt the guest OS (e.g. in response to a timer) and thus introduce extra latency into potential measurements. In practice, this turned out to be a rare event, but nonetheless it was easy to detect because it produced extremely large messaging latency measurements. These were discarded in order to filter out this type of interference.

Execution of the RDTSC instruction alone has some inherent overhead. Rather than allowing this overhead to artificially increase the duration of cycle measurements, back-to-back TSC instructions were executed in order to calculate the time it takes the processor to perform zero work. The result was non-zero and was subtracted from each messaging measurement in order to correct for this effect.

A final concern for benchmarking messaging performance is that processors might engage in frequency scaling. Without a stable clock frequency, it is very difficult to determine how cycles (as measured by the TSC) correspond to the passage of time. For all tests, frequency scaling was disabled because of these concerns. This includes Intel's Turbo Boost feature, where a core can be overclocked as long as thermal conditions allow for it.

5.2 Messaging Performance

Two experiments were conducted in order to determine the performance of fos messaging. First, the messaging latency of fos was compared to Linux system call latency. Second, the effect of message size on latency was measured. Both experiments used a message echo operation to simulate typical communication patterns. A message echo operation works by sending a fixed-size message back and forth between a pair of communicating processes.

Since fos dedicates entire cores to system services, it is assumed that each process will run on a separate core. The latency of the cache coherency protocol can vary greatly depending on which cores are used. Because these benchmarks aim to fully understand the effects of heterogeneity, measurements were taken with processes running on two cores from the same chip (shared L3 cache) as well as processes running two cores from different chips (no shared caches.)

5.2.1 Performance of Messaging for System Calls

The fos microkernel supports system calls much like other OSes. However, it would be unfair to use fos microkernel system call latency as a basis of comparison. In fos, most OS services are accessed through message passing instead of trapping into the kernel. Therefore, a roundtrip message between an application and a fos fleet member is the closest fos equivalent to a conventional trap-and-enter system call.

When evaluating the performance of system calls it is important to measure the raw overhead imposed by the platform without including operations specific to a particular type of request. For example, system call benchmark results would be less useful if they included the latency of disk access along with the overhead of entering and exiting the kernel. To this end, only the performance of null system calls, or system calls that perform few or no operations, was measured. For Linux, the `GET_PID` system call was used because it performs merely a few memory references. Linux supports a variety of system call mechanisms. On the AMD and Intel processors tested, the most performant method was to use the `SYSENTER` instruction, and thus this method is used in all experiments. For fos, a message echo operation was performed to simulate a null fos system call. The size of the message was kept small (32 bytes.), representing an operation that receives a typical set of arguments and immediately gives back a return value.

Table 5.1 shows latency measurements for fos echo operations and Linux system calls. In general, user messaging was considerably faster than kernel messaging. Moreover, Linux system call latency had similar performance to user messaging, especially on the Intel processor. That is to say, user messaging is sufficiently fast to perform a system call on a different core from the application without negating the cache locality benefits of such a placement policy. It also appears that the Intel Xeon E5530 has moderately better cache coherency and trap performance than the AMD Opteron 6168.

Linux was measured as a guest inside a Xen DomU, the same platform as fos. However, moderately better Linux System call performance is expected on a baremetal

		AMD Opteron 6168	Intel Xeon E5530
fos Userspace	shared cache	1236 (463)	618 (114)
	no shared cache	1381 (509)	984 (619)
fos Kernelspace	shared cache	4785 (1107)	3117 (271)
	no shared cache	5501 (914)	4010 (243)
Null Linux System Call		874 (120)	726 (251)

Table 5.1: A latency comparison of null fos system calls (roundtrip messaging) and null Linux system calls. Measurements are in nanoseconds (σ).

host. This is primarily because Xen requires a change in the page table root in order to trap into the guest kernel, resulting in a greater number of TLB lookups [23].

5.2.2 Latency for Different Message Sizes

Another dimension of messaging performance is the ability to handle different message sizes efficiently. While smaller message sizes tend to represent most procedure calls, large message sizes are an important workload for fos too. Specifically, they are typical of I/O operations such as reading and writing disk blocks or sending and receiving network packets.

Figure 5-1 shows the performance of fos messaging for a spectrum of different message sizes. The results indicate that user messaging approaches about half the latency of kernel messaging for increasingly large message sizes. Certainly one reason is that user messaging has lower latency for each message sent because it avoids entry into the kernel. Perhaps the most significant reason, however, is that user messaging allows the receiver to begin receiving pieces of the message, in cache-line sized chunks, before the full message has been enqueued. This allows the sender and receiver to work in parallel to perform the data transfer. Of course, kernel messaging could be modified to operate in a similar fashion, but such an implementation would be more difficult because the kernel must cope with multiple concurrent writers.

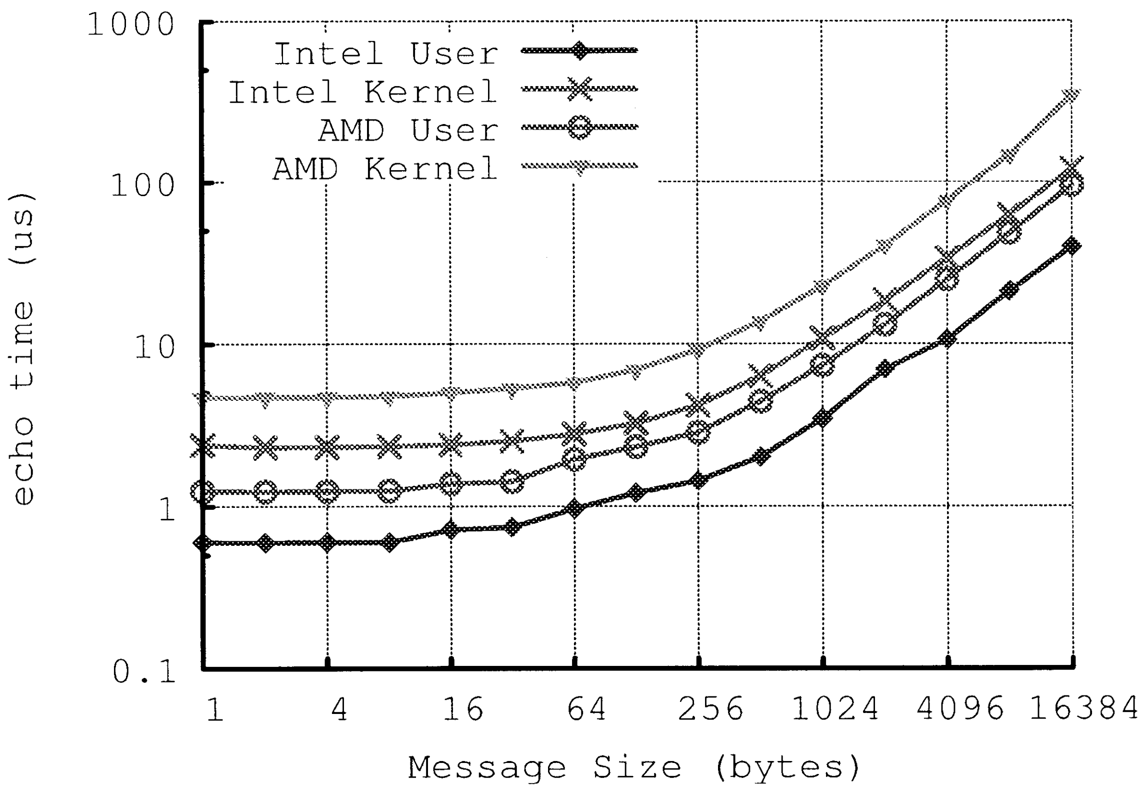

Figure 5-1: Messaging latency for a variety of message sizes. Results are shown for both kernel and user messaging on each of the tested processor architectures.

5.3 Dynamic Transport Selection

Figure 5-2 shows fos dynamically switching from kernel to user messaging in response to mailbox activity. These results were obtained on the Intel processor using cores without a shared cache. The transition threshold occurred after the fourth message. The figure further illustrates that although creating a shared memory channel costs over 40,000 cycles, the cumulative cycles spent on user messaging, including channel creation, become less than the cumulative cycles spent on kernel messaging after about six messages.

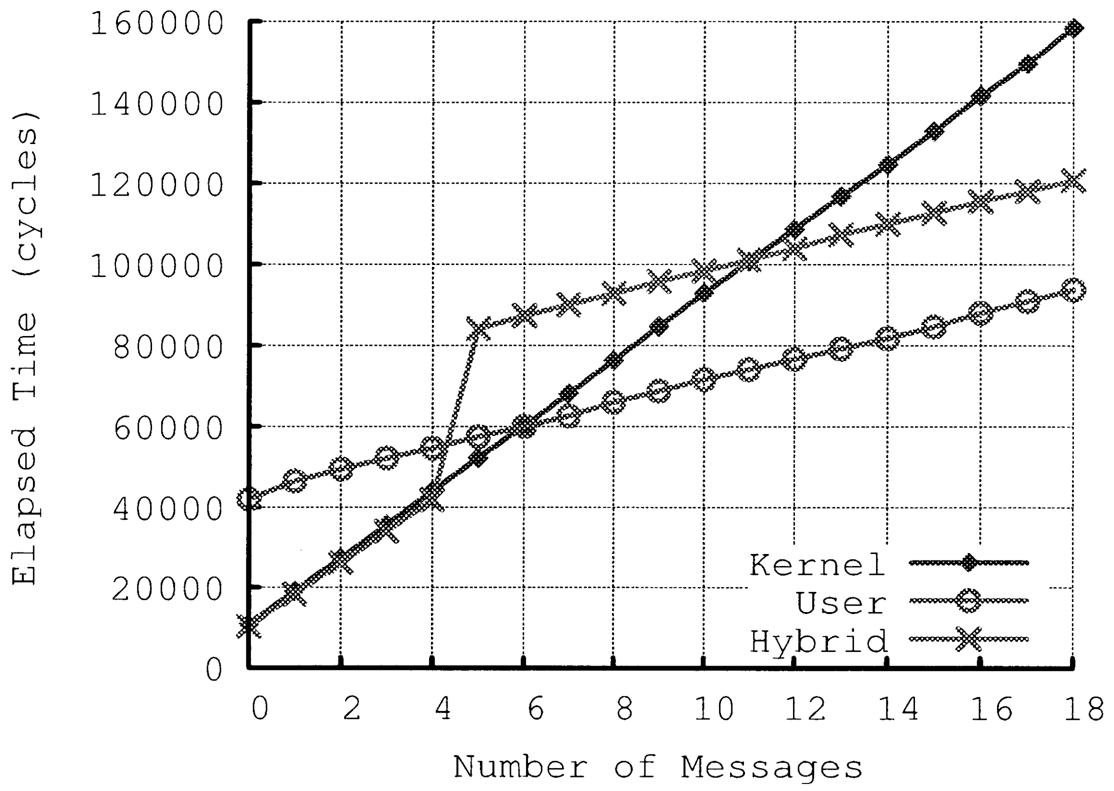

Figure 5-2: The cumulative cycles spent for different messaging policies. Hybrid messaging illustrates fos dynamically switching from kernel to user messaging. As a basis for comparison, cumulative cycles are also shown for the policy of only using user messaging and the policy of only using kernel messaging.

By dynamically switching from kernel to user message, the cost of setting up a shared memory channel is avoided for one-off messages while still getting lower latency messaging for sustained message requests. Thus, a competitive messaging approach represents the best of both options.

5.4 Is Lower Latency Possible?

fos user messaging has performance that is comparable to system calls under a Xen Linux guest. Nonetheless, an interesting question is whether even better performance is possible. To explore this inquiry, a stripped-down IPC implementation was developed for the purpose of determining the minimum possible latency. The complete code of the implementation is included in appendix A. It simulates a system call by passing a typical collection of register state to a server process. The server process then executes the request and gives back a return value. The implementation is extremely restrictive, supporting only cache-line sized data transfers and only a single request at a time. As such, it is not a full message passing system and thus would not be suitable for fos. Nonetheless, it does serve to establish a lower bound on achievable latency.

Careful management of cache lines is the key to the implementation's good performance. Specifically, the client packs the system call arguments into a cache-line sized and aligned region of shared memory. When the all of the arguments are ready to be sent to the server, a flag is set in the last word of the cache line, indicating that the requester is finished. At the same time, the server constantly polls the last word in the cache line, waiting for the flag to be set. When the flag is detected, the server knows that all of the arguments are available. It then executes the request, and writes over the first word of the cache line with a return value. Finally, to indicate to the client that the return value is ready, it clears the flag in the last word of the cache line. After making a request, the client polls the last cache line in a similar fashion to the server and waits for the flag to be cleared. When the flag is cleared it reads the return value and the request is complete.

The performance of this procedure is optimal because under typical conditions it results in only two cache misses, one on each core. Specifically, the client places the line in the modified state when it writes the arguments and the flag bit. The line then gets transfered to the server, resulting in the first cache miss. Similarly, the server places the line in the modified state when it writes the return value and clears the flag bit. The line gets transferred back to the client, resulting in the second cache miss.

	Latency in Cycles (σ)
fos user messaging	1479 (272)
fast minimal IPC	160.8 (66)

Table 5.2: A comparison of fos messaging latency (for a small message echo) to a stripped-down fast IPC implementation. Tests were run on two cores sharing a cache on the Intel processor.

Table 5.2 compares the performance of the fast IPC implementation with fos user messaging. In both cases, the tests were run on the Intel processor with a shared cache between the cores. The results indicate that there is considerable room to improve performance, with the fast IPC implementation being around nine times faster.

fos user messaging is slower for a variety of factors. First, a buffer must be allocated for each message in order to support arbitrary length messages as implemented in the fos messaging API. Second, since all messages are one-way, fos user messaging must use two seperate cache lines to send and receive data between a pair of processes, resulting in twice as many cache misses. Finally, because name lookups must be performed (in a hashtable-based local memory cache,) a small number of cycles is added to each message send. Despite these extra cycle penalties, fos's user messaging achieves very good performance, especially considering that it is a more flexible and complete system. Nonetheless, it may be possible to improve fos user messaging performance further by using some of the techniques in the fast IPC implementation.

Chapter 6

Conclusion

This thesis presented a message passing system for the fos OS. The system supports two kinds of messaging transports. The first is kernel messaging, which has higher latency but lower setup costs. The second is user messaging, which has lower latency but higher setup costs. It was demonstrated that a dynamic transport selector can switch between the two mechanisms, achieving good latency for frequent communication while avoiding setup costs for one-off messages. Moreover, the fos proxy service was introduced to provide seamless messaging access to different machines on the network. This makes it easier to construct applications that use cores across several machines.

Another contribution of this thesis is an analysis of the performance ramifications of running the OS on a different core than the application. It was shown that fos has the potential to better utilize the cache resources of multicore chips, making it a preferable OS design for the increasingly large core counts of emerging chips. The performance of fos messaging was evaluated, and in the case of user messaging, it was measured to be about as fast as Linux system calls on the same platform. Thus, the cache benefits of running an OS on a different core from the application are not mitigated by the cost of sending requests between cores.

6.1 Future Work

There is great potential to further improve the performance of messaging in fos. First, the fast IPC implementation in appendix A demonstrates that faster messaging, with the right restrictions, is possible on cache coherent processor environments. An interesting challenge would be to incorporate some of the ideas in the fast IPC implementation into fos. One option might be to have an RPC stub generator encode function arguments directly into shared memory, making it more natural to use a fixed size message buffer. Of course, hardware accelerated messaging could also be a promising avenue for improving message performance. Thus, porting fos to a different hardware platform, such as Intel's SCC could be a good way to investigate the performance benefits of new kinds of hardware support.

Another area worth exploring is process scheduling and its related implications to power management. Currently the fos messaging system spends a lot of time polling mailboxes in order to determine if new messages have arrived. It would be more efficient if processes yielded control of their cores to the kernel after waiting for a period of idleness. However, this could negatively impact performance in some situations. One idea might be to have a fos system call that passes a hint to schedule a process when a message has been sent to it. A similar technique was developed for URPC messaging, primarily for single core processors [27].

Appendix A

Fast IPC Implementation

This section presents source code for a minimal optimized IPC implementation designed for cache coherent shared memory. Each request results in only two cache invalidations.

Listing A.1: syscall.h

```
#define cpu_relax() __asm__ ("pause":::"memory")
#define barrier() __asm__ __volatile__("":::"memory")

enum {
  REG_RAX = 0,
  REG_RDI,
  REG_RSI,
  REG_RDX,
  REG_RCX,
  REG_R8,
  REG_R9,
  NUM_REGS
};

struct syscall_packet {
  int64_t regs[NUM_REGS];
  int64_t ctl;
} __attribute__((packed)) __attribute__((aligned(64)));

#define CTL_READY 0x1

static inline void set_ready(volatile struct syscall_packet *pkt)
{
  pkt->ctl |= CTL_READY;
}

static inline void clear_ready(volatile struct syscall_packet *pkt)
{
  pkt->ctl &= ~(CTL_READY);
}

static inline int is_ready(volatile struct syscall_packet *pkt)
{
  return (pkt->ctl & CTL_READY);
}
```

Listing A.2: client.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "syscall.h"

#define N 5000

static char *shm;

static inline unsigned long rdtscll(void)
{
  unsigned int a, d;
  asm volatile("rdtsc" : "=a" (a), "=d" (d));
  return ((unsigned long) a) | (((unsigned long) d) << 32);
}

static unsigned long calculate_tsc_overhead(void)
{
  unsigned long t0, t1, overhead = ~0UL;
  int i;

  for (i = 0; i < N; i++) {
    t0 = rdtscll();
    asm volatile("");
    t1 = rdtscll();
    if (t1 - t0 < overhead)
      overhead = t1 - t0;
  }

  printf("tsc overhead is %ld\n", overhead);

  return overhead;
}

static void run_test(void)
{
  unsigned long tsc, result;
  unsigned long overhead = calculate_tsc_overhead();
  int i;
  volatile struct syscall_packet *pkt = (struct syscall_packet *) shm;

  for (i = 0; i < N; i++) {
    tsc = rdtscll();

    pkt->regs[REG_RAX] = 1;
    pkt->regs[REG_RDI] = 10;
    pkt->regs[REG_RSI] = 20;
    barrier();
    set_ready(pkt);

    while (is_ready(pkt))
      cpu_relax();

    if (pkt->regs[REG_RAX] != 30) {
      printf("message send failed\n");
      exit(1);
    }

    result = rdtscll() - tsc - overhead;
    if (result < 10000)
      printf("%ld\n", result);
  }
}

int main(int argc, char *argv[])
{
  int shmid;
  key_t key;

  key = 5679;

  if ((shmid = shmget(key, 4096, 0666)) < 0) {
    perror("shmget");
    exit(1);
  }

  if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
  }

  run_test();

  return 0;
}
```

Listing A.3: server.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "syscall.h"

static char *shm;

static int64_t add_nums(int64_t a, int64_t b)
{
  return a + b;
}

static void run_server(void)
{
  volatile struct syscall_packet *pkt = (struct syscall_packet *) shm;

  while (1) {
    while (!is_ready(pkt))
      cpu_relax();

    if (pkt->regs[REG_RAX] != 1) {
      printf("message receive failed\n");
      exit(1);
    }

    pkt->regs[REG_RAX] = add_nums(pkt->regs[REG_RDI],
                pkt->regs[REG_RSI]);
    barrier();
    clear_ready(pkt);
  }
}

int main(int argc, char *argv[])
{
  int shmid;
  key_t key;
```

```
    key = 5679;

    if ((shmid = shmget(key, 4096, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    run_server();

    return 0;
}
```

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[2] A. Agarwal, R. L. Sites, and M. Horowitz. Atum: a new technique for capturing address traces using microcode. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 119–127, 1986.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh 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.

[4] Nathan (Nathan Zachary) Beckmann. Distributed naming in a factored operating system. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2010.

[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, 2008.

[6] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[7] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8:37–55, February 1990.

[8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:175–198, May 1991.

[9] Dileep Bhandarkar and Jason Ding. Performance characterization of the pentium pro processor. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 288–297, February 1997.

[10] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2008.

[11] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI 2010: Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation.*

[12] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[13] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international*

symposium on Computer architecture, ISCA '96, pages 78–89, New York, NY, USA, 1996. ACM.

[14] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 120–133, 1993.

[15] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 189–202, New York, NY, USA, 1993. ACM.

[16] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 347–348, New York, NY, USA, 2010. ACM.

[18] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[19] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[20] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance*

Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[21] P. V. Mockapetris. Domain names - implementation and specification, 1987.

[22] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261 –270, 2009.

[23] Jun Nakajima and Asit Mallick. X86-64 xenlinux: Architecture, implementation, and optimizations. In *Proceedings of Linux Symposium*, 2006.

[24] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[25] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the simos approach. *Parallel Distributed Technology: Systems Applications, IEEE*, 3(4):34 –43, 1995.

[26] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*. ACM.

[27] R.C. Unrau and O. Krieger. Efficient sleep/wake-up protocols for user-level ipc. In *Parallel Processing, 1998. Proceedings. 1998 International Conference on*, pages 560 –569, aug 1998.

[28] VMware Inc. *VMCI Sockets Programming Guide.*

[29] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, September 2007.

[30] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 3–14, New York, NY, USA, 2010. ACM.

[31] David Wentzlaff, III Gruenwald, Charles, Nathan Beckmann, Adam Belay, Harshad Kasture, Kevin Modzelewski, Lamia Youseff, Jason E. Miller, and Anant Agarwal. Fleets: Scalable services in a factored operating system. Technical Report MIT-CSAIL-TR-2011-012, MIT CSAIL, March 2011.

[32] Donald Yeung, John Kubiatowicz, and Anant Agarwal. Multigrain shared memory. *ACM Trans. Comput. Syst.*, 18:154–196, May 2000.

[33] Lamia Youseff, Dmitrii Zagorodnov, and Rich Wolski. Inter-os communication on highly parallel multi-core architectures.