Scalable Memory Management Using a Distributed Buddy Allocator

by

Kevin Modzelewski

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

June 2010

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2010

Certified by . . . .
Anant Agarwal
Professor, CSAIL
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Scalable Memory Management Using a Distributed Buddy Allocator

by

Kevin Modzelewski

Submitted to the Department of Electrical Engineering and Computer Science
May 20, 2010
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The recent rise of multicore processors has forced us to reexamine old computer science problems in a new light. As multicores turn into manycores, we need to visit these problems yet again to find solutions that will work on these drastically different architectures.

This thesis presents the design of a new page allocator algorithm based on a new distributed buddy allocator algorithm, one which is made with future processor architectures in mind. The page allocator is a vital and heavily-used part of an operating system, and making this more scalable is a necessary step to build a scalable operating system.

This design was implemented in the fos [34] research operating system, and evaluated on 8- and 16-core machines. The results show that this design has comparable performance with Linux for small core counts, and with its better scalability, surpasses the performance of Linux at higher core counts.

Thesis Supervisor: Anant Agarwal
Title: Professor, CSAIL

Acknowledgments

I would like to thank David Wentzlaff and Anant Agarwal for their supervision and guidance on this project. David's knowledge and experience proved invaluable in dealing with the issues that arose over the course of this research, and Anant's ability to see the big picture was always able to keep me on the right track.

This thesis would not have been possible without the fos project, and I'd like to thank everyone else in the group as well for their help and advice, including Charles Gruenwald, Nathan Beckmann, Adam Belay, Harshad Kasture, Lamia Youseff, and Jason Miller.

Finally, I would like to thank my parents for their support, without which I would not be where I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen the proliferation of multicore systems, with mainstream 12-core processors available [3] now, and research and specialized processors with 64 or more cores [2, 20, 27, 32]. The growth of multicore processors has kept processor aggregate performance on the same trend that, until now, has been sustained by increases in single-stream performance. This performance, however, comes at the cost of being harder for programmers to use. Software can no longer continue expecting easy speedups by using processors with higher clock speeds; instead software must be specifically designed to use the new type of power that multicore processors offer.

The operating system is one piece of software that has no choice but to make use of this parallelism. If the operating system cannot scale, then none of the applications running on it can either. There are many components of an operating system that present parallelization challenges, one of which is the memory management subsystem.

The memory management system is itself composed of smaller pieces; one of the more important ones is the *page allocator*. The page allocator is responsible for managing the set of allocated or free pages in the system, and is used for most memory management functions.

This thesis presents a new technique for scalable memory management by using a distributed buddy allocator as the page allocator. Recent research [31] has shown that the page allocator in Linux does not scale to the number of cores currently available, and that replacing this component can increase the scalability of the system. The contribution of this thesis is the design, implementation, and evaluation of a novel distributed buddy allocator. Results show that the new allocator is competitive with Linux for small core counts but achieves better scalability, to the point that it is faster for the number of cores one may find in a machine today.

This work was done as a part of the larger fos [34] project. fos is a new operating system that represents a comprehensive redesign of the fundamental operating system architecture, and aims to replace all traditional operating system services with scalable variants. This page allocator is one such service.

The design of fos presents many difficulties and opportunities for doing scalability research. A second goal of this project was to help evaluate the decisions made as part of the fos project.

The structure of this thesis is as follows. Chapter 2 describes the problems that the memory management system solves, along with the current state of the art. Chapter 3 goes into the design motivations and choices behind the parent project, fos. Chapter 4 details the new design presented in this thesis. Chapter 5 presents results obtained by implementing and running this design. Chapter 6 identifies future areas for study, and chapter 7 concludes.

Chapter 2

Background

2.1 Page allocators

Page allocators play a vital role in modern operating systems. RAM is one of the key resources in computer systems, and the page allocator is responsible for the efficient management of RAM. Whenever a kernel requires more physical memory – such as to spawn a new process, load data off a disk, or expand the heap of a current process – the kernel requests a range of free pages from the page allocator. Page allocation is the process of managing the physical pages of memory in the machine, keeping track of which are free or in use, and allocating them to the requesting process. The allocator needs to distribute only free pages to requests, and it also needs to be able to reallocate pages that have been freed. Due to other design considerations, the page allocator returns *contiguous* set of pages. The page allocator needs to efficiently respond to these requests, to keep the system running quickly.

There are a range of criteria for allocator algorithms, some of which are less relevant in a page allocation context:

Performance One of the key metrics of any allocator is how quickly it can return requested ranges. A key component of good performance is high scalability.

Fragmentation There are many types of fragmentation. *Internal* fragmentation is the space that is wasted because a larger region was returned than necessary.

This is less important for a page allocator, since the requester (the kernel) is free to use small ranges and stitch them together to form a larger range. *External* fragmentation is the condition when there is free memory, but it exists in disjoint ranges which cannot be combined into a single larger range. *Distributed* fragmentation is when there is free memory, but no single process knows enough about the global state to coalesce them. This has a similar solution to the problem of *passively-induced false sharing*, which is encountered in dynamic memory allocators. *Blowup* [8] is similar to distributed fragmentation, but happens when the free memory becomes imbalanced between the different processors. In dynamic memory allocators, this results in higher memory usage than required; in a page allocator, it results in lower memory efficiency.

Locality, False Sharing Modern memory allocators optimize for high locality and low false sharing. These are considerations for shared-memory dynamic memory allocators, though, which do not apply to a page allocator.

There are a range of allocator algorithms that make different tradeoffs between these features. Many contemporary operating systems, including Linux, choose to use buddy allocators for their page allocators due to their high speed. Buddy allocators can quickly provide ranges of pages where the number of pages is a power of two, and can quickly split and coalesce ranges. The suffer, though, from relatively high internal fragmentation; operating system kernels, though, are able to work around this limitation.

2.1.1 Buddy Allocators

Buddy allocators can be thought of as a binary tree, where the leaves represent individual pages of memory, and are marked "free" if the corresponding page is free. Inner nodes represent the range of pages corresponding to the union of their children, and are marked free only if both of their children are (and thus all of their descendants). When servicing a request, the buddy allocator first rounds up the number of pages to a power of 2. It then locates a range with that many pages that is marked free.

(This can be found efficiently by maintaining a linked list of all free ranges at each level.) It then marks that range allocated, and returns the corresponding range. To service a request to free some pages, the buddy allocator first marks the range free. Then, if that range's sibling (the range's "buddy", hence the name) is also free, the allocator coalesces the two ranges by marking the parent free. If the parent's buddy is also free, it recursively continues the process.

Due to their simple design and good performance, buddy allocators have been the algorithm of choice for kernel page allocators. Similar to many parts of the operating system, however, buddy allocators were chosen for their single-stream performance, and work is needed to provide high performance as the number of cores increases. The original method of making the page allocator work in a multi-cpu environment was to place a big lock around the buddy allocator. While this successfully prevented incorrect behavior, it also prevented increased page allocation performance from additional cores. There have been efforts to make the page allocator system more scalable, including making a concurrent buddy allocator, and using per-cpu caches to avoid taking out the big lock.

There are many concurrent buddy allocator structures, such as the one presented in [22]. It uses per-freelist locks, which means that any two requests for different powers of two can proceed in parallel. While effective, this limits the parallelization to the number of freelists in the allocator, which is a fixed number as the number of cores grows.

Linux has addressed this problem by using per-cpu free lists. Each core maintains a small list of free pages, and when that cpu requests a single page, it services the request from the cache. In order to reduce memory waste due to pages sitting in the caches, Linux only caches single-page ranges. Recent research [31] has shown that this approach works well for small numbers of cores, but does not scale. A more promising direction is motivated by recent work on a closely-related subject: dynamic memory allocation.

2.2 Dynamic memory allocation

Dynamic memory allocation, one of the common tasks in computer programming [35], shares many things in common with page allocation. In both problems, the algorithm must allocate contiguous ranges, and be able to free them to be reallocated later, while obtaining good speed and memory efficiency. In fact, using a buddy allocator is a possible method of dynamic memory allocation. Dynamic memory allocation, though, prioritizes memory efficiency, which is the opposite tradeoff that a buddy allocator makes, and buddy allocators are now rarely used for dynamic memory allocation. Since the problems are similar, though, ideas from good dynamic memory allocation can be used to created better page allocators (and vice-versa).

Dynamic memory allocation has similarly had to be approached in new ways as the number of cores in systems grows. There has been much research into making parallelizing dynamic memory allocation, and efforts fall into a few main categories:

- Big locks around sequential algorithms. This is the simplest method, though it does not allow additional performance by adding more cores.

- Concurrent structures, such as [22] and [21]. These work similarly to the concurrent buddy allocator: by using finer locks. Again, though, these suffer from having scalability limited by the number of concurrently requested sizes.

- Purely-private heaps, such as the ones in CILK [1] and the STL [28]. In these schemes, processors use the same heap for all allocations *and* frees. This leads to very high performance, but can lead to problems in certain use cases where memory is freed from different processors than it is allocated on.

- Local heaps plus a global heap, such as Hoard [8], Streamflow [26], and others [16, 17, 23, 25, 30]. These use local heaps to serve as many requests as possible, but use the global heap when necessary. This is similar to the way that Linux uses a per-cpu cache of free pages, but uses the global buddy allocator when necessary.

Allocators that predominantly use their local heaps, such as Hoard and Streamflow, achieve good performance and scalability, while addressing the issues with purely-private heap allocators such as CILK and the STL. This thesis makes the following contributions on top of these allocators:

- Application to a buddy allocator, instead of a free-list allocator. This makes a tradeoff that is more favorable to being a kernel page allocator.
- Extension to settings where there is no global heap. The global heap is what allows the different local heaps to remain balanced; with no global heap, processes must rely on a different rebalancing mechanism.
- Extension to architectures without shared memory. These allocators generally use shared memory to coordinate between the different heaps (and manage the global heap). The allocator presented here uses only message passing, and is able to work on machines without coherent shared memory.

The global heap must be eliminated for a few reasons. First, it is a source of contention between the different processes. Contention on the global heap increases linearly with the number of processes, which eventually becomes a problem. This can be addressed by increasing the size of allocations from the global heap, but this leads to linear per-thread memory overhead, instead of constant. Second, without shared memory, having a global heap becomes more difficult. It is possible to have a single process manage the global heap, but the design needs to be complicated to avoid having that process be a single point of failure for the allocator.

The natural next step in this evolution is to use an all-distributed design, taking ideas from the area of highly-scalable distributed systems. In this kind of design, all of the memory would be managed in private heaps, but the private heaps would have the ability to communicate with each other to balance the system. This achieves the same fast-path performance, because most operations are able to be served from the local heap. The slow path – when the local heap is unable to service a request – is now more scalable, since any other local heap would be able to provide more memory, instead of the single global heap, which would eventually become a bottleneck. Streamflow [26] uses this idea to make remote deallocations fast; this design proposed in this thesis extends it to rebalancing in general.

The problems of page allocation and dynamic memory allocation, though different in many important ways, have very similar algorithms. Both classes of problems are now running into the importance of scalability in addition to single-stream performance, and the work in both areas is heading towards an all-distributed design.

Chapter 3

fos

3.1 Motivation

fos [34], a factored operating system, is a current research project exploring a new operating system design for future computer architectures. The rise of multicore and cloud computers present difficulties for traditional operating system designs, which did not take such systems into account. fos aims to redesign the way the operating system works, taking scalability as a first order design constraint. In fos, all system services must scale to meet the demand of hundreds or even thousands of cores. It is within this context that the page allocator presented in this thesis was designed; this chapter describes the previous work that has gone into fos.

The motivation for fos is driven by recent trends in processor design. Moore's law predicts an exponential rise in processor transistor counts, and until recently this extra processing power has translated into higher single stream performance. Recently, though, physical constraints have put a stop to this avenue. Instead, processor manufacturers have turned to increasing the number of cores on the same die. This change means that the total computational power of new chips continues to grow at the same rate, but also that software developers face new challenges, as this power becomes more and more difficult to utilize. In particular, recent research [10] has shown that modern operating systems have difficulty using this additional power.

In fact, it is often the case that there is more processing power available than the developer can effectively utilize, and this is only becoming more common. This means that the system is CPU-bound, but there are cores that are idle. This leads to the *abundant cores* assumption: that in the future, we will have more cores available than can be used effectively by the user, and priority shifts to using the idle cores to run user applications faster.

The problem is that traditional operating systems are not designed to be able to make this shift. They were designed at a time when multiple processing units were rare and esoteric, and the resulting fundamental design reflects this. Given the new architecture landscape of today, it makes sense to create a new operating system from scratch, that is designed to address these problems directly.

There are two components to building a scalable operating system: the first is providing a scalable abstraction to user applications, and the other is efficiently implementing that abstraction. The new interfaces that fos provides are described in 3.2; in order to provide efficient implementations, we must rewrite large parts of the operating system, such as the page allocator.

3.1.1 Multicore

There are additional problems that arise in the process of designing and construction a scalable operating system. The hardware, itself, is difficult to scale as the number of cores increase. One particular place this shows up is in the cache-coherent shared memory system. Cache-coherent shared memory becomes more and more expensive to implement and use as the number of cores increases, and for this reason many new unconventional architectures do not support it [19, 20, 32].

Instead, many new architectures offer an efficient message passing mechanism. This is much easier for the hardware to provide, but it offers new challenges for the programmer, along with some new opportunities. It means that, for better or for worse, all applications written on top of fos are more similar to current distributed applications than parallel programs.

Future operating systems must also deal with additional problems that do not occur in current systems. In the future, it is likely that individual cores will experience the same classes of errors that internet-scale systems do today, such as single-node failure, lossy communication, and network partitions.

Additionally, the networking topologies of chips will likely become more heterogeneous. Already, there is a large difference in communication latencies between cores [10], depending on the location of the cores. This suggests that spacial locality will be a primary performance factor for future systems for future systems.

3.1.2 Cloud

At the same time that processors have increasing numbers of cores, developers are using increasing numbers of machines in their systems. Currently, this distributed systems work must be done at a level above the operating system. Given the similar problems presented by multicore and the cloud [34], though, the operating system is able to use the same mechanisms to help with both.

With no shared memory, messaging as the communication mechanism, independent processing, independent fault model, heterogeneous network topologies, and an increasing amount of networking with other processors, future processors look more like distributed systems than they do current parallel processors. fos uses this insight, and borrows many ideas from distributed systems research to inspire the design of an operating system for these processors.

3.2 Design

While fos is designed for future hardware, it is currently implemented for x86_64 processors. Although shared memory is available, it is not used in any of the operating system services to preserve compatibility with future hardware. It is implemented as a hypervirtualized kernel, running on top of a Xen [6] hypervisor. While using Xen frees us from worrying about device compatibility, it unfortunately changes the performance characteristics of the "hardware". Xen is structured as a privileged "domain 0" (*dom0*) virtual machine (VM), with several guest "domain U" (*domU*) VMs. The dom0 is usually a Linux operating system, and fos runs as a domU.

3.2.1 Fleets

fos is a microkernel based operating system that uses a fundamentally different system architecture from conventional monolithic kernels. In fos, each process is considered to be a *server*, and the terms are used interchangeably. Each server is assigned to a core, and there is only limited time-multiplexing on each core, since it is assumed that each server will make maximal use of the core's resources. The operating services in fos assume that there is no shared memory in the system, so the base communication mechanism is message passing. Thus the basic architecture of fos is similar to what one might find in a distributed system: many independently-running servers that perform service in response to requests from their clients, which may be other servers.

Corey [10] has shown the benefit of splitting operating system services from user applications, and fos uses this principle by factoring the operating system into function-specific services. Furthermore, each service is provided by a distributed, co-operating *fleet* of servers. In systems with thousands of cores, a single core is likely to not be capable of serving requests from all the other cores, and thus needs to be parallelized.

Figure 3-1 shows what the layout of fos servers may look like on a manycore chip. Each core runs a single server on top of a microkernel, and the servers are spatially scheduled onto the different cores. A *libfos* library layer provides compatibility with legacy applications by converting POSIX syscalls into fos syscalls, or messages to appropriate servers.

The fleet-based design of fos has many benefits. It includes the benefits of Corey: less cache contention between the kernel and the user applications, and better control of contended resources such as a network interface card. Additionally, the parallel nature of fos fleets allows for greater performance. In particular, page mapping performance is highest with three cores mapping pages, and fos allows the service to spawn multiple servers in the fleet to meet this.

One unavoidable difficulty with this model is that it is harder to change the amount of resources devoted to a specific service. In a traditional multicore environment, this

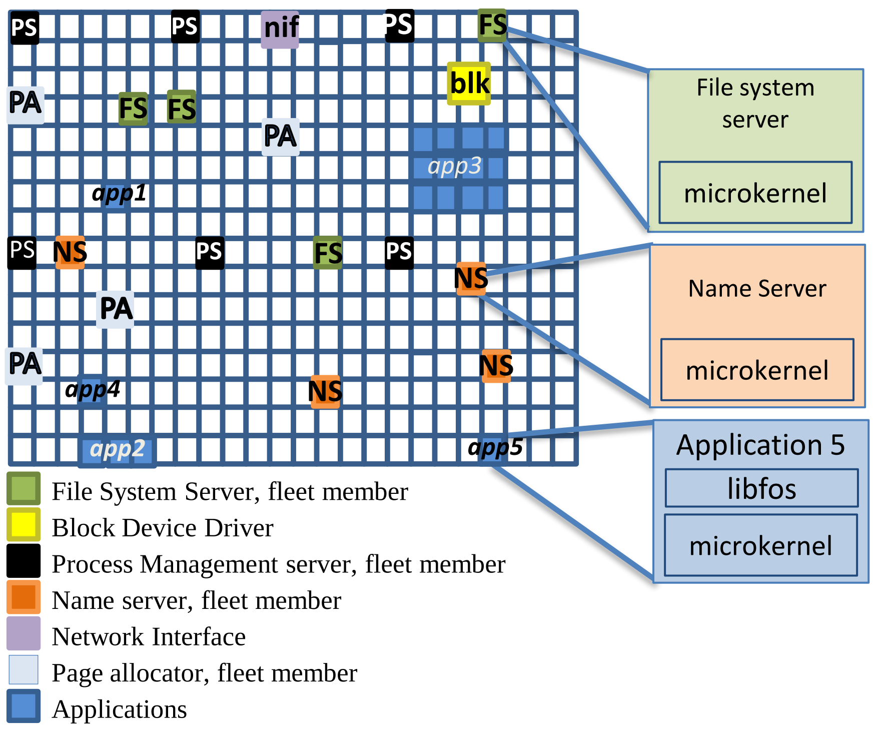

Figure 3-1: A diagram of the layout of servers in fos. Reproduced from Figure 1 in [33].

redistribution be accomplished by assigning more or fewer time quanta to a process. In fos, however, each process is generally the only thing running on its core. In this model, changing the amount of resources means changing the number of cores assigned to a fleet, which, as opposed to changing the amount of time it has, is a non-transparent procedure. Since we expect demands on operating services to be highly *elastic*, services must be designed to adapt to this. This means that services must be prepared to gain or lose servers from their fleets.

3.2.2 Messaging

Another way that fos aims to improve the scalability of the operating system design is by redesigning the interfaces provided to the programmer, and inter-process communication is a good example of this. Since processes are no longer able to communicate with shared memory, all communication happens over fos-provided messaging channels. fos uses this to provide the same interface for inter- and intra-chip communication, and transparently switching between the two as necessary.

All communication in fos happens via mailboxes. fos allows processes to create and register mailboxes with the system, and then wait for messages to arrive on them. Mailboxes are registered via the name server, which provides a name lookup service similar to DNS. The addresses returned by the name server are opaque to user applications, which allows fos to transparently choose the message transport mechanism, based on the destination of the message.

fos currently implements three separate messaging transports: one that uses the kernel, one based off of URPC [7, 9], and one that uses the network interface. Each serves a separate purpose: the kernel messaging requires little set up, but is somewhat slow; URPC is much faster, but requires some setup and management overhead; using the network stack allows messages to be transported between machines. Since the inter-machine transport uses the same interface as the local messaging, the programmer simply writes messaging code, and fos determines if the destination is on the same machine or not, and chooses an appropriate transport. This is important, given fos's multi-machine focus.

3.2.3 Scalability Support

Message transport transparency is only one of the ways in which fos helps programmers write the distributed systems needed to achieve good scalability. It also provides a new programming model, with an associated runtime, that makes it easier to write these services. The programming model is centered on a cooperatively-threaded event dispatch paradigm, which allows the programmer to write straight-line code and worry about the functionality of their system rather than the exact mechanics. It also allows the registration of asynchronous handlers, which is made possible by the fact that servers run on their own dedicated cores.

fos also provides an RPC code generation library that produces code for the dispatch library to use. This allows the programmer to write code that looks like a normal function call on both sides, and the library will take care of serializing the function call, sending the request, waiting for the response, and deserializing the response.

fos provides helpful mechanisms for other hard distributed systems problems, such as state sharing, elasticity, and load balancing. Unfortunately, they were in active development at the time of this thesis, and were not available for use.

Additional details of the system architecture and programming model can be found in [34] or [31].

3.3 Related work

fos is similar to, and draws from, many other works. It extends the microkernel designs first implemented in Mach [4] and L4 [24]. There have been many other efforts to produce scalable operating systems: Tornado [15] and its successor K42 [5], Disco [11] and Cellular Disco [18], and Hive [12] have all focused on scaling operating systems to larger numbers of cores. Unlike fos, though, they target shared-memory machines, where the number of cores is in the dozens.

More recent efforts include Corey [10], which investigated the idea of using dedicated cores to do specific operating system tasks. They show, for instance, that by using a dedicated core to handle the network stack, the system is capable of much greater network performance. fos uses this idea heavily, and takes it farther by using multiple cores to provide each service. Barrelfish [7] is a similar project to fos, investigating operating system design for manycore systems that may not have shared memory, using a client-server model. fos also includes a cloud focus, and research into parallelizing individual operating services (such as this thesis).

3.4 Implications

Doing this page allocator research in the context of the future-hardware operating systems research has a number of implications stemming from the fact that the page allocator must use the interfaces provided by the operating system. The first is that these interfaces do not make full use of today's hardware, because they assume that today's hardware abstractions will be too expensive to provide in the future. This puts an extra constraint on the way the page allocator can be designed; in particular, the page allocator cannot use shared memory. The second is that since these interfaces are designed for future hardware but running on today's hardware, they do not perform as well as today's interfaces on today's hardware. This means, as well, that since the page allocator is optimized for future hardware, it is likely to not provide as good performance on today's hardware as algorithms optimized for today's hardware; in particular, it assumes the existence of fast message passing primitives, which do not exist on today's commodity processors.

It does mean, however, that the page allocator is able to make the same assumptions about future hardware as fos. In particular, the design in this thesis assumes that there are semi-idle cores that are available to run the page allocator server, which follows from fos's abundant cores assumption. This allows the page allocator to assume that its servers run on their own cores, and thus are able to do extra computation without delaying application processes.

Being written inside fos also means that the page allocator is able to make use of the programming model that fos provides. One of the secondary goals of the project was to evaluate how much help fos provides in writing scalable systems, and I found that the programming model made it significantly easier to write the page allocator.

These constraints also differentiate a fos page allocator from other existing algorithms, as well as dynamic memory allocation. In a system with no shared memory, it is impossible to have a global shared heap, as described in [8]. It is harder, in general, to have a global view of the system; this makes it so that the page allocator is not able to efficiently keep track of the total number of free pages in the system, and thus it does not know when the system is actually out of memory or not. The page allocator must also not be implemented with a design that contains a single point of failure, because it is possible that in a manycore chip with thousands of cores, single cores can fail at a time. This leads to a fully-distributed and -decentralized design.

Chapter 4

Design

There are two main components to the memory management subsystem: a service for getting ranges of free physical pages, and a service for mapping those returned pages into a requested page table. The page allocator is discussed in section 4.1, and the page mapper in 4.2.

4.1 Page Allocator

The main component of the memory management service is an efficient, scalable page allocator. The page allocator allows its clients to request contiguous ranges of pages, as long as the number of pages is a power of two and less than a certain value. The page allocator is responsible for keeping track of which pages are currently allocated and which are free, and only returning free pages as the result of an allocate request. To do this performantly, we use a new *distributed buddy allocator* algorithm.

This algorithm is designed to be used as a page allocator for the types of systems that fos is aimed at. It could be adapted to create a dynamic memory allocator instead, though several of the assumptions of fos are at odds with the problem of scalable dynamic memory allocation. Primarily, since fos assumes there is no shared memory (and that lack is one of the main motivations of this new design), the design tradeoffs may not make sense in a system with shared memory, which parallel dynamic memory allocation implies. The ideas, however, should still be transferable.

4.1.1 Distributed Buddy Allocator algorithm

The distributed buddy allocator can be thought of as the natural next step in the evolution of scalable allocators, as described in chapter 2. It is designed as a fleet of cooperating buddy allocator servers, each of which maintains its on private heap of free pages. Each free page is known to be free by one and only one server; this means that that server is free to allocate its pages because it knows no other server will, so that each server is able to execute independently of the others.

The fleet starts by spawning a single coordinator server. The coordinator is distinguished only in fleet setup, and afterwords behaves exactly like the other servers. The coordinator node then spawns as many new servers as needed to get the desired number of servers, including itself. The coordinator then informs all of the spawned processes of each other, starts the fleet, and drops its coordinator role.

The coordinator also, in this process, assigns ranges of pages to an *owner*. The owner of a range of pages is the server that is in charge of managing that range, though at any given point in time a page may be managed by any server. The page space is split into ownership ranges that are multiples of the largest possible allocation range. This means that any range of pages that could be allocated or freed will entirely belong to a single order. Each server manages its list of owned pages using a conventional buddy allocator.

Each server is also able to manage pages that it does not own. This can happen when a client returns a range of pages to a server that does not own them, or when a server needs to rebalance pages from another server (see section 4.1.2). The total number of "foreign" pages that a server is managing is bounded, though (section 4.1.3), and is much smaller than the number of pages that each server owns. To prevent high memory overhead, each server uses a *sparse* buddy allocator structure to manage unowned pages. Although the sparse structure is less memory efficient per-page, the total memory usage is smaller than using a conventional dense buddy allocator for all of the pages in the system.

The ownership map is currently assigned statically, for ease of implementation,

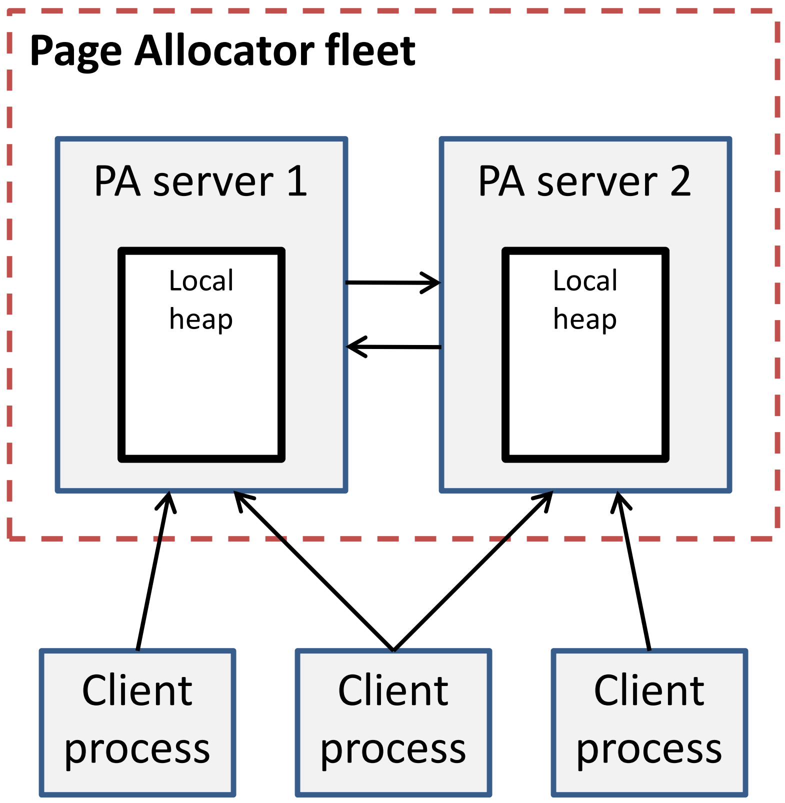

Figure 4-1: Architecture of the page allocator fleet

and for performance. It is possible to add new servers to the fleet, but the only way for these new servers to gain pages is through the rebalancing mechanism. Additionally, changing ownership of pages is not currently supported, and this represents an interesting avenue for future research.

The allocator servers accept requests (via messaging) for ranges of pages, and any allocator is able to handle any request. Each allocator tries to execute all requests against its local heap. When a server receives a request free'ing a range of pages, it adds them to its local heap, rather than trying to return them to where they were originally allocated from, or the owning server in case they are different. This saves an extra round trip whenever a client free's a range of pages to a separate server than they were allocated from, since if needed the pages can be sent back later in the background.

Figure 4-1 shows the high-level architecture and communication patterns of the page allocator fleet, for an example fleet of two servers. Each server manages its own local heap, which only it is able to access. The servers make use of other local heaps by sending messages to the servers owning the other heap. These two servers form the page allocator fleet, and collectively implement the distributed buddy allocator structure.

Figure 4-1 also shows three example clients. Generally, each client will send all of its messages to a single page allocator server, but it is possible for a single client to send messages to multiple servers. Due to the design of the page allocator fleet, though, the servers can transparently handle this situation, though with slightly less efficiency.

4.1.2 Rebalancing

In the common case, the server is able to satisfy its requests with pages in its local heap. In this case, the operation is exactly like a serial buddy allocator, except that each server is able to proceed in parallel, because each knows that its heap is strictly private.

When usage becomes imbalanced, though, and some page allocator servers give out many more pages than others, a server may run out of pages in its heap before the others. In this case, the depleted server will request a large range of pages from a nearby server to refill its heap. This is similar to the way that Hoard [8] will refill its local heap by allocating out of the global heap, but with a few important differences.

The first difference is that there is no global heap to refill from. Instead, a server must talk to one of its peer servers. Since there is no shared memory in fos, a depleted server is not able to directly access the local heap of another server, but must instead send a request to the owner of that heap.

Second, as will be described see in section 4.1.3, moving pages around too much causes excessive fragmentation. To prevent this, borrowing pages is seen as a temporary measure to maintain performance, rather than a permanent reassigning of pages from one server to another.

Third, due to the architecture of fos, the distributed buddy allocator is able to do this asynchronously, and thus not introduce any latency into user applications. Since each server runs on its own core, the server is free to use its idle cycles however it wants, including getting more pages. When a server notices that it is about to run out of pages, it will asynchronously start requesting more pages from other servers. Because buddy allocators efficiently allocate and free large regions at a time, the rebalancing process only takes a single message, in which a large number of pages are moved.

The combination of fast and scalable common case performance, along with efficient background rebalancing, means that the distributed buddy allocator tends to have very good performance. There are, however, some additional aspects to deal with beyond allocation speed.

4.1.3 Challenges

The distributed buddy allocator is designed to give fast, scalable, performance, but there are some other problems that arise. Buddy allocators generally have good external fragmentation, and in a page allocation context, their relatively-poor internal fragmentation is not an issue. However, in a distributed context, buddy allocators can suffer from excessive blowup due to *distributed fragmentation*.

Since the servers do not always return free pages back to the server they were allocated from, it is possible that all the memory in the system is free, but it is scattered between the different local heaps in such a way that none of it can be coalesced, which is a case of distributed fragmentation.

To combat this, each server implements a high-page threshold, in addition to the low-page threshold under which it will try to acquire more pages from peers. When the server has more pages than its high threshold, it will try to return pages to their owners. This makes it so that neighboring pages eventually make it back to the same server, so that they can be coalesced into a larger range.

When a server receives a free request for a range of pages that it does not own, and it is currently above the high threshold, it will immediately forward the free request to the owning server. (It does this in the background, though, so that it can immediately return to the client.) This means that a server can never have more foreign pages than the high threshold number of pages.

The value of the difference between the high threshold and the low threshold is an important tuning parameter in this system, controlling the balance between rebalancing cost and memory overhead. By making these two thresholds close together, the system will have to work hard to keep all servers in that range, but it puts a tight bound on the amount of memory that can be wasted to distributed fragmentation. Using a larger difference, in comparison, makes it easy for the servers to stay in that range, but does so by possibly allowing a correspondingly larger amount of distributed fragmentation.

There are number of strategies for the order in which to return non-owned pages to their owners, such as returning the smallest range first, the largest range first, or the oldest/newest range first. These all worked well in practice, but without real-life traces to test on, it is not clear *a priori* which ones are better.

Another challenge faced by such a distributed system is not having a global view of the system. In particular, no server knows the total number of free pages over all of the servers. This is particularly important in low-memory situations, since servers no longer know when the system is completely out of memory. It would be possible to keep track of this quantity, such as through a gossip-based protocol, but this requires more overhead, especially in an environment where the set of page allocator servers is allowed to change.

Instead of sending extra messages to determine how many pages the other servers have, servers return an "out of memory" error code whenever they have had no memory for a period of time. Since servers are asynchronously trying to acquire more pages from other servers, if a server has been unable to acquire more pages then its peers are also low on pages. Inability to acquire more pages, then, is a good indicator of the system being out of memory.

Using the same signal – the number of pages that a peer server was able to return – the servers form an estimate of the number of pages free in the system. They use this to adjust their high- and low-amount thresholds, to be proportional to the number of pages that they would have if the system was completely balanced. This avoids thrashing issues when the average number of pages per server moves too much; if there were fixed thresholds, any time this average went above the high threshold or below the low threshold, the system would be saturated with rebalancing traffic. Using dynamic thresholds, this is avoided.

4.1.4 Alternative designs

There were a few alternative designs that were considered, but subsequently rejected.

Client retry

Rather than have the servers do page rebalancing themselves, it is possible to signal to the client that there is no memory available on that server. It would be the client's responsibility to find another page allocator server, and request pages from it.

There are a few advantages to this method: it gives the client complete control over the exact retry strategy, allowing for different request types. It is simpler to implement, especially if there is an automatic load balancing mechanism.

There are more disadvantages though: when a server runs out of pages, it has reduced the number of pages in the fleet that are able to satisfy requests, lowering throughput. If there is only one server with pages left, it should give those pages out to the other servers so that they can help satisfy demand. Also, when the server rebalances pages, it has the option of requesting more pages than it needs, reducing the number of needed rebalance operations. The server also has the option of doing this asynchronously, without introducing any latency into client operations. The server also can keep track of which other servers still have pages left, and only send requests to those.

Master server

Rather than using a completely decentralized design where there are no distinguished servers in the fleet, it is possible to have a distinguished "master" server that plays the role of the global heap. In this model, each server only maintains a small cache of

Number of cores	Throughput (pages per million cycles)
1	237
2	306
3	207
4	157
5	136
6	116
7	110

Table 4.1: Page-mapping throughput as a function of cores

pages, to satisfy the next few requests. When a server needs more pages, it requests some more from the master server. When it has too many, it returns some to the master server.

This has the benefit of simplicity and ease of adding more servers. It makes it harder to deal with distributed fragmentation, since it is no longer clear who is currently managing which regions of memory. It also suffers from the same problem as global heaps in dynamic memory allocation: as the load on the master/global heap grows, the contention and latency for accessing it will grow as well. Rather than build a hierarchical system, we chose to go with a fully-distributed design that has the potential for much farther scalability.

4.2 Page Mapper

The final component of the memory subsystem is the service that actually maps pages into page tables. Because fos is written as a Xen guest operating system, this involves doing a hypercall to the Xen hypervisor.

Unfortunately, we are not able to map pages very scalably on our machine. Table 4.1 shows the total throughput of a set of cores constantly trying to map pages, as quickly as possible. Each testing process sits in a loop mapping a random page.

As one would expect, the total throughput increases when you add a second core. Afterwords, however, performance actually *decreases*. This type of bottleneck is a common occurrence, and fos provides a straightforward solution: factor out the service. Hence, a page mapping service.

This service is possible due to the microkernel nature of fos, and is feasible due to its capability system. The fact that fos is a microkernel means that it provides the mechanism for mapping pages, but allows userland programs to control the policy. It uses capabilities to ensure that only the page mapping service is allowed to execute the privileged operation of mapping pages into arbitrary page tables.

The job of this service is to limit the number of cores that are simultaneously trying to tell the Xen hypervisor to change page tables. By limiting this number, the number of cores mapping pages is decoupled from the number of cores wanting pages to get mapped. This means that the total page-mapping throughput can be kept at the peak of table 4.1, rather than falling off as the number of cores is increased.

Chapter 5

Results

The presented design has been implemented and evaluated inside the fos operating system. It has been tested with a synthetic workload designed to stress the page allocator specifically, which directly access the page allocator rather than go through the microkernel. It has not been fully integrated as the system page allocator.

There were two separate machines used for testing: a machine with dual quad-core Xeon E5530 2.4 GHz processors, and a machine with four quad-core Xeon E7340 2.4GHz processors; both were running a Xen 3.2 hypervisor with a 64-bit Linux 2.6.26 dom0. For any comparisons made in this chapter, all results in the comparison were taken on the same machine. The different comparisons were run on different machines (due to the limited availability of a 16-core machine), and may not be directly comparable to each other.

5.1 Implementation

The memory management system is implemented as a fleet of cooperating servers, which communicate via message passing. The servers are written using the fos programming model, which allows for straight-line code and RPC-like interfaces.

The memory management system is broken into a page allocator service and a page mapper service. While logically separate, for performance reasons they are run inside the process. This allows us to cut down the number of required cores (since we do not yet have an abundant-core system to test on), without changing the performance characteristics.

fos makes the implementation of this design straightforward, though there were a number of pitfalls. It is important to not saturate the chips communication bandwidth with rebalance traffic, so it was necessary to add exponential backoff to the rebalance requests on failure. It is also important to set the thresholds correctly; otherwise servers can thrash and repeatedly trade the same range of pages back and forth.

The implementation was evaluated using a set of page allocator clients, which directly talk to the page allocator servers. These clients independently simulate a workflow of allocating and releasing pages, slowly increasing the overall number of in-use pages. The throughputs shown here represent the average time per allocation request, including the amortized time to later free the pages.

These results are compared against results from a Linux test designed to be as close a match as possible. In the Linux test, multiple processes allocate a large region of memory, and proceed to touch a single byte in each page. Linux does not actually allocate or map pages when they are first requested, but rather when they are first used; this is why the test must touch a byte in every page, to force the page to actually be allocated and mapped. The throughputs shown for Linux represent the average time to touch the first byte, including process setup and teardown time. Linux maps each page on demand (and does not speculatively map the next page), so touching the page for the first time corresponds to allocating and mapping it. Setup and teardown time is included to capture the effects of freeing the pages, since Linux may reclaim them only when needed again. The overhead due to starting and killing a new process is negligible relative to the length of the test.

In the fos test, none of the allocation requests are batched or pipelined, to match the way the Linux test works. The frees, however, were both batched and pipelined. The free requests were batched by placing up to 128 separate page ranges of the same order into a single request. Since there is a large per-byte cost to messaging, batching more than 128 pages at a time does not increase performance measurably. The free requests were also pipelined, where multiple free requests were allowed to be in-flight

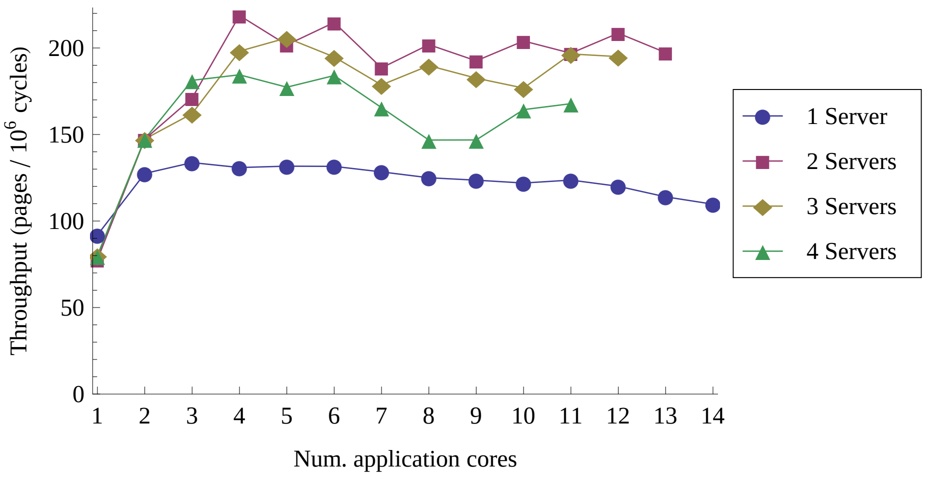

Figure 5-1: Page allocator performance, varying the number of servers

at the same time. This improved performance by allowing the page allocator client and servers to simultaneously communicate in both directions.

5.2 Performance

Figure 5-1 shows the performance of the implementation of this design. The test is constructed by having a variable number of clients simulate a synthetic workload, and requesting the servers to map pages into their address spaces, and was run on the 16-core machine. After the pages are mapped, the clients zero the page and ask for a new page.

The different series in this figure represent runs with different numbers of servers, as the load is increased by adding more clients. This figure shows that increasing the number of servers helps at first, then has no effect, than decreases the overall throughput. Adding clients has a similar effect: it takes up to four clients to fully saturate the fleet, at which point adding more clients does not increase performance, but rather it drops off slowly.

Figure 5-2 illustrates the overhead due to mapping the pages. The first line is the speed of the system when no pages are mapped or touched. For reference, the "ideal" speed is plotted as well, which is assuming linear speedup off of the one-core

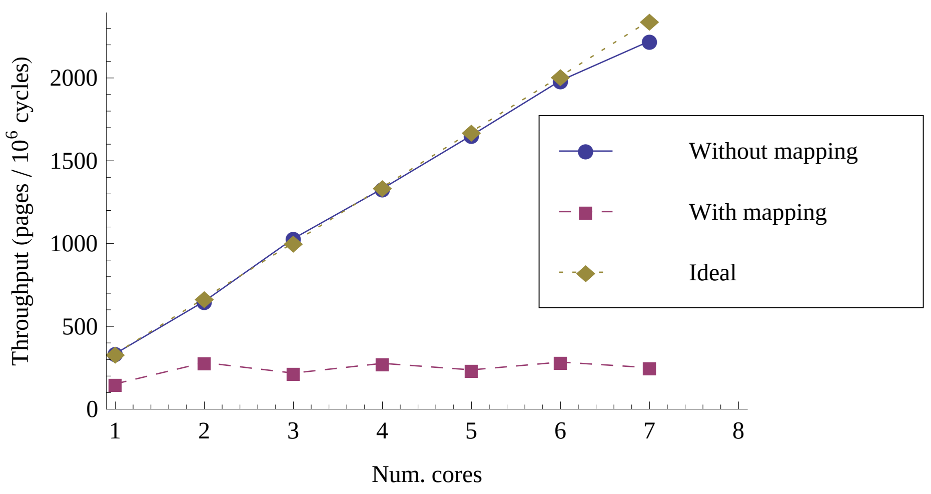

Figure 5-2: Memory management performance, with and without mapping pages

case. The lower line shows the performance when the memory management system also maps the page. Since the only difference between these two lines is whether or not the page is mapped (in neither case is it touched after being mapped), this shows the large overhead of mapping pages.

The allocate-only line gets quite close to the ideal scaling line. This is because the buddy allocator is sufficiently fast that most of the time is spent in the messaging layer. Since the messaging layer scales almost linearly, the buddy allocator (without page mapping) does too.

This figure, along with table 4.1, shows that the page allocator algorithm itself scales quite well, whereas page mapping quickly becomes bottlenecked by hardware and the Xen hypervisor. Figure 5-3 shows the difference in page mapping speed in Linux when running on raw hardware versus running as a Xen guest. This shows that the Xen hypervisor adds some overhead to page mapping, but the hardware architecture still limits overall page mapping throughput.

Ultimately, we suspect that future hardware will support parallel page table updates more efficiently, especially if there is no shared memory. Additionally, we suspect that running fos on raw hardware, rather than in a virtual machine, will also increase the speed. Unfortunately, while running on today's hardware, the memory system is largely bottlenecked by the hardware architecture.

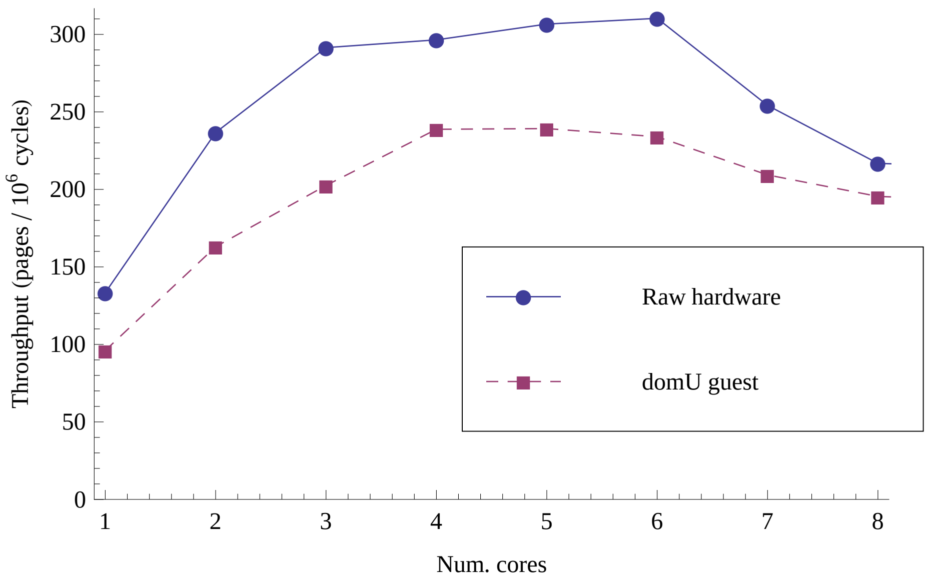

Figure 5-3: Linux page-mapping throughput when run on raw hardware vs. run as a Xen domU guest.

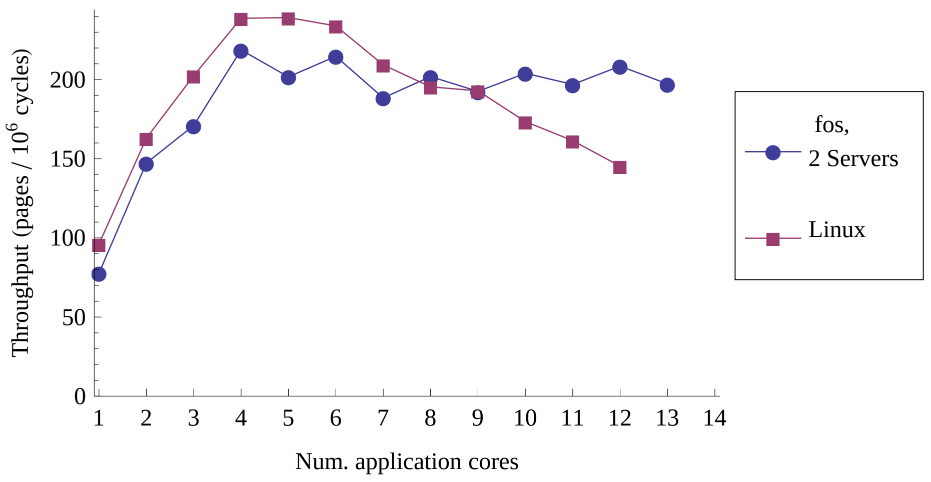

Figure 5-4: Page allocator performance of fos vs. Linux

Number of clients	1	2	3
Bad layout	130	259	386
Good layout	166	334	452
Performance increase	28%	29%	17%

Table 5.1: Page allocator throughput for different process layouts

Figure 5-4 shows a comparison of the fos memory system (using 2 servers) versus Linux, graphed the same way as figure 5-1.

This figure shows that Linux has somewhat higher throughput for small numbers of cores. This is to be expected, since this is what Linux is optimized for. Potential reasons for this include that Linux avoids having to do an inter-core message for each page map, and that Linux maps the pages on the same core that they are used, which could lead to cache benefits. The fos numbers are competitive with the Linux numbers, though, staying within 10-20%.

For higher core counts, however, the fos memory management system is better able to maintain performance. As was shown in table 4.1, as the number of cores trying to map pages increases the overall throughput of the system goes down; due to the design of Linux, as the number of allocating processes increases, the number of page-mapping cores must also increase. In fos, however, we are able to cap the number of page-mapping cores to the number that gives maximum throughput, and keep it there even as the number of allocating processes increases. This is why fos is able to maintain peak performance as the number of processes increase, as opposed to Linux.

Also in this figure, one can see the lack of load balancing in the current page allocator system. In the test, each client picks a single server to communicate with. When there are two servers and an even number of clients, the load is distributed evenly between the two servers. When there are an odd number of clients, though, the load is distributed unevenly, and the total throughput is slightly lower.

Table 5.1 shows the effects of the differences in inter-core communication latencies. The first line represents the performance (in page allocations per million cycles) with the worst case assignment of processes to cores, which maximizes average message delay. The second line shows the performance for a process layout that has been optimized to minimize message delays.

This shows that process layout is an important factor in performance. The difference is as almost 30% for small numbers of cores, and decreases for more cores. The reason for this decrease is that for small numbers of cores, all cores can be very close together, but by going above 4 cores, some must be placed on the second processor, which increases the average message delay.

Since fos does not currently migrate processes between cores to improve locality, the all other tests mentioned in this section all used the optimized process layout.

Chapter 6

Future Work

There are many interesting questions that were outside the scope of this research, many of which are general distributed systems research questions.

Rebalancing and Allocation strategies There are several options for rebalancing that were not explored. The current design decides to rebalance solely based on the total number of pages in the local heap; there are other factors, though, that could lead to better rebalancing decisions. For instance, if there are many requests for large ranges of pages, it may make sense to consider the sizes of the ranges in the local heap. Additionally, the balance between owned and unowned pages could be a helpful factor.

The rebalancing mechanism could also be made more sophisticated. As mentioned in section 5.1, the rebalancer uses an exponential backoff strategy to reduce rebalancing communication. This works well when rebalancing is a infrequent operation, which it usually is. If rebalancing is more frequent, the servers should send multiple rebalance requests out in parallel to improve rebalancing throughput.

The allocation strategy treats owned and unowned pages the same: when a server has both owned and unowned pages in its heap, it does not allocate one preferentially to another. It may make sense to always allocate owned pages first, or unowned pages first, or use some other factor (such as locality information) to make a smarter decision. The allocator also does not consider fragmentation when allocating pages:

it would be possible to pick a particular page range based on, for example, how fragmented its parent ranges are.

The performance of these tradeoffs, though, is heavily influenced by the request pattern made to the servers. Unfortunately, we do not have realistic usage traces, so we cannot meaningfully evaluate the decisions. (The fos kernel currently always allocates a single page at a time.)

Fault tolerance It is suspected that future processors will exhibit more failure modes than current ones. With hundreds, or thousands, of cores on a single die, it is possible that an individual core may fail, but leave the rest operational. Many of the services in fos are being designed to be completely tolerant to these kinds of stop-faults; this page allocator design did not incorporate that as a first order constraint, and so does not have true fault tolerance. Because it has no single point of failure, though, it has a limited amount of fault containment: if a node goes down, only the pages currently being managed by that node will be lost. Future research could examine the feasibility of ensuring that no pages become permanently unusable.

Elasticity As mentioned, the distributed buddy allocator has a limited ability to tolerate elasticity. It is possible to add new servers beyond the original set that it started with, but it is not that efficient, and it is also currently not possible to remove servers from the original set. This is because the page allocator fleet statically partitions the page range to different owners. Having some sort of dynamic update mechanism for ownership information would solve this issue, at a cost of greater complexity and greater overhead.

Using a design more similar to DHTs such as Chord [29] and Dynamo [14] could be one possible way of addressing these issue. These systems are designed to provide elasticity, and to a more limited extent fault tolerance, but the performance of the design would necessarily suffer. Elasticity handling is an active area of research for fos.

Raw hardware fos currently only runs as a Xen guest on current x86 processors. This means that it is limited by the performance characteristics of both systems. In particular, both the Xen hypervisor and consumer x86 processors have poor page mapping scalability. Doing a bare-metal port of fos, especially on a specialized processor, would put more stress on the page allocator rather than the page mapper, and allow for more investigations in that area.

Locality With the increasing heterogeneity of processor interconnects, cores are increasingly seeing non-uniform memory access (NUMA) latencies to the different DRAM banks. This means that, for the page allocator, certain pages are better to return, because they will be closer to the requesting client. Additionally, when running across multiple machines, only certain physical pages will be relevant. The page allocator should be augmented with location information, so that it knows which pages to return preferentially to which clients.

Chapter 7

Conclusion

This thesis presents a new distributed buddy allocator structure, and uses it to construct a more scalable memory management system for fos. Despite not being designed for current hardware, the resulting memory management system is able to stay competitive with Linux under synthetic stress tests, and has better throughput at higher core counts. While the performance of the memory management system is still dominated by hardware architectural overheads, results show that the presented design is well-poised to take advantage of future hardware architectural changes.

Bibliography

[1] Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, November 1994.

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

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

[4] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, June 1986.

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

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew 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.

[7] 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.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

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

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

[12] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 12–25, 1995.

[13] Juan A. Colmenares, Sarah Bird, Hentry Cook, Paul Pearce, David Zhu, John Shalf, Steven Hofmeyr, Krste Asanovic, and John Kubiatowicz. Resource management in the tessellation manycore os. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, 2010.

[14] Giuseppe DeCandia et al. Dynamo: amazon's highly available key-value store. In *SOSP '07*, pages 205–220, New York, NY, USA, 2007. ACM.

[15] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael 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, February 1999.

[16] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/ wmglo/malloc-slides.html.

[17] Google. Google performance tools. http://code.google.com/p/google-perftools/.

[18] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel 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.

[19] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March-April 2006.

[20] 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.

[21] Arun K. Iyengar. *Dynamic Storage Allocation on a Multiprocessor.* PhD thesis, MIT, 1992.

[22] Theodore Johnson and Tim Davis. Space efficient parallel buddy memory management. Technical Report TR92-008, University of Florida, Department of CIS, 1992.

[23] Per-Ake Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the First International Symposium on Memory Management*, October 1998.

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

[25] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, June 2004.

[26] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, 2006.

[27] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat 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.

[28] SGI. The standard template library for c++: Allocators. http://www.sgi.com/tech/stl/Allocators.html.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and K. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *In Proc. SIGCOMM (2001)*, 2001.

[30] Voon-Yee Vee and Wen-Jing Hsu. A scalable and efficient storage allocator on shared memory multiprocessors. In *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, June 1999.

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

[32] 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(5):15–31, September 2007.

[33] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Harshad Kasture, Lamia Youseff, Jason Miller, and Anant Agarwal. Parallel operating system services in a factored operating system. Unpublished manuscript submitted for publication, 2010.

[34] 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 ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[35] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, 1995.