dPool: A Distributed Data Structure for Factored Operating Systems

by

David Wentzlaff

B.S., University of Illinois at Urbana-Champaign (2000)
S.M., Massachusetts Institute of Technology (2002)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

Author . . . . . .
Department of Electrical Engineering and Computer Science
September 8, 2011

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

Certified by . . . . . .
Srinivas Devadas
Professor
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Chair, EECS Committee on Graduate Students

dPool: A Distributed Data Structure for Factored Operating Systems

by

David Wentzlaff

Submitted to the Department of Electrical Engineering and Computer Science
on September 8, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Future computer architectures will likely exhibit increased parallelism through the addition of more processor cores. Architectural trends such as exponentially increasing parallelism and the possible lack of scalable shared memory motivate the reevaluation of operating system design. This thesis work takes place in the context of Factored Operating Systems which leverage distributed system ideas to increase the scalability of multicore processor operating systems. fos, a Factored Operating System, explores a new design point for operating systems where traditional low-level operating system services are fine-grain parallelized while internally only using explicit message passing for communication. fos factors an operating system first by system service and then further parallelizes inside of the system service by splitting the service into a fleet of server processes which communicate via messaging. Constructing parallel low-level operating system services which only internally use messaging is challenging because shared resources must be partitioned across servers and the services must provide scalable performance when met with uneven demand.

To ease the construction of parallel fos system services, this thesis develops the dPool distributed data structure. The dPool data structure provides concurrent access to an unordered collection of elements by server processes within a fos fleet. Internal to a single dPool instance, all communication between different portions of a dPool is done via messaging. This thesis uses the dPool data structure within the parallel fos Physical Memory Allocation fleet and demonstrates that it is possible to use a dPool to manage shared state in a factored operating system's physical page allocator.

This thesis begins by presenting the design of the prototype fos operating system. In the context of fos system service fleets, this thesis describes the dPool data structure, its design, different implementations, and interfaces. The dPool data structure is shown to achieve scalability across even and uneven micro-benchmark workloads. This thesis shows that common parallel and distributed programming techniques apply to the creation of dPool and that background threads within a dPool can increase performance. Finally, this thesis evaluates different dPool implementations and demonstrates that intelligently pushing elements between dPool parts can in-

crease scalability.

Thesis Supervisor: Anant Agarwal
Title: Professor

Thesis Supervisor: Srinivas Devadas
Title: Professor

Acknowledgments

This work and my graduate career could not have been possible without the help of many great friends, colleagues, and mentors. I would like to start by thanking my parents for being huge supporters of me. They raised me in a very nurturing, warm, and loving environment. They have also always encouraged me to pursue education.

My advisor Anant Agarwal has taught me what it means to be a great researcher. He has guided me in both academic pursuits and collaboration with others. Completing my thesis will be sad as I know it will mean an end of an era of learning everything that I can from Anant. Hopefully I will still be able to learn from him in my future career.

I would like to thank Saman Amarasinghe and Martin Rinard who have always been willing to give me advice when I have needed it. They have served as great sounding boards. Martin provides great clarity when trying to cut through dicey issues in both life and academia.

I was assigned Srini Devadas as my academic advisor by chance, but I am so happy for that good fortune. He has provided me a great mix of professional career guidance and has always been eager to talk to me about technical issues I have had.

I am indebted to the other members of my thesis committee, Frans Kaashoek and Robert Morris. They have taught me the most about what it means to research operating systems and their feedback has been invaluable.

During my early years at MIT, I had a great time learning about how to build real computer systems in the Raw group. This team was the best team that I have ever worked with. We could have interesting technical discussions, build real working systems, and hang out socially. I learned so much from this group and our counterparts in Saman's group. I would like to call out a few of my mentors in this group. Michael B. Taylor was both a mentor and is a great friend. I still remember the many hours we spent discussing different computer architectures while playing Ms. Pacman and eating greasy pizza. Matt Frank taught me how to be an academic and how to read and review papers critically. Walter Lee taught me everything I would

ever need to know about bridge and then some, but also taught me how to engineer complex software systems. Jason E. Miller and I have spent countless hours in the lab together. He is a great friend and made those hours more fun.

I would like to thank the team at Tilera for enabling my development as a computer architect. I want to especially call out Bruce Edwards and Carl Ramey who taught me that computer architecture wisdom is worth more than its weight in gold. I am thankful that John F. Brown III and Richard Schooler really care about their teams, much more than just as numbers. I would like to thank Ken Steele, Ethan Berger, Chris Jackson, John Amann, and John Zook for being great friends during my Tilera experience.

The fos team has been filled with a new crop of stellar students and researchers. This thesis was only possible with their tremendous support.

Bill Thies has been a wonderful friend and partner in crime in graduate school. Without his friendship, grad school would have been much less exciting. Chris Batten has been a wonderful friend in graduate school, and he is always willing to listen to my crazy ideas. I would like to thank Chris for his help in preparing for the post graduate school experience.

I would like to thank my high-school physics teacher John Taska for getting me excited about science. His passion as a teacher made a big difference in encouraging at least one pupil.

My fiancé Cassandra has been my muse and a huge support during the last few years of my graduate career. I would like to thank her for always being supportive, giving me a hug in the challenging times, and for being a beacon of sunshine when I have been down.

This work has been funded by Quanta Computer, DARPA, AFRL, NSF, and Google.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis focuses on techniques and data structures which enable the creation of scalable operating systems for current and future multicore processors. fos [73, 75] is a Factored Operating System designed to provide scalable performance on multicore processors. fos works toward this goal by utilizing distributed system techniques inside of an operating system's core functionality. One of the key ways that fos provides scalability is by forcing the operating system programmer to explicitly think about and manage communication. It does this by organizing the OS as a set of processes which only communicate via explicit message passing. While making communication explicit has advantages such as preventing implicit sharing which can limit scalability, requiring explicit communication can make managing logically shared state, which may be easy in a shared memory environment, challenging. This thesis addresses the challenge of managing shared state for one particular shared state usage pattern, namely a unordered collection of elements. This thesis develops dPool, a distributed data structure with unordered multiset semantics, which provides a concurrent interface across OS processes to shared state. dPool thereby provides much of the ease of shared memory programming while internally only utilizing messaging. dPool's interface is designed such that it can store different sized elements to facilitate it being used by different fos services. This thesis focuses on the performance and scalability of dPool being used inside of the fos Physical Memory Allocation service.

1.1 fos

The design of fos is motivated by anticipated future multicore architectures and computer architecture trends. I believe that for the foreseeable future, Moore's law [48] will continue to afford chip designers more usable transistors for a fixed price. It is likely that this increasing number of transistors will be turned into additional processor cores in a single, tightly coupled computer system. The fos project focuses on how to construct operating systems for these highly concurrent architectures without the operating system becoming the performance bottleneck. Because Moore's law dictates that the number of transistors and processor cores will be increasing at an exponential rate over time, the fos project focuses on not only on how to construct an operating system for a fixed large numbers of cores, but how to construct an operating system which can continue to scale out, thereby providing more OS throughput as the number of cores grows in future systems.

The era of multicore processors has introduced the challenge and opportunity of exponentially increasing core count which OSes have not traditionally had to tackle. The fos work assumes that the traditional approach of monolithic OS design will have challenges meeting the new scalability demands of exponentially increasing core count. In the fos project, we have identified several of the problems facing monolithic OS design on future multicore architectures including: reliance on shared memory locks, inability to control OS and application working set aliasing in caches, and reliance on shared memory for implicit communication. In contrast to traditional monolithic OSes, fos controls its core-to-core communication by using explicit message passing between different portions of the OS. The choice to use messages in fos is driven by multiple factors. First, the fos team believes that future multicore processors will likely not have scalable global coherent shared memory. It is possible that future multicores will have regions of shared memory, poor performing shared memory, or no on-chip coherent memory due to the hardware cost. Second, many research multicore processors have explicit hardware messaging and fos would like to take advantage of this mechanism. Although this thesis focuses on multicore processors, by having a

message based design, fos has been extended across clusters and clouds of multiple computers. Finally by using messaging, the fos system developer can be extremely cognizant of when communication is occurring and program accordingly. Even though fos holds the systems programmer to a high standard by requiring message passing to be used for internal communication, fos does not hold the application user to the same high standard as fos can execute user applications which use multiple threads and shared memory, provided that the underlying hardware supports shared memory.

In order to increase the available parallelism inside of the operating system, Factored Operating Systems, such as fos, begin by factoring an operating system by the service provided. Each of the services provided by the OS is further parallelized into a fleet of cooperating server processes which collectively provide a single operating system service. These operating system servers and applications do not share processor cores and in fact, each server process and application is bound to a different core in a multicore system to reduce contention on the capacity of a single core's cache. Applications communicate with system services only via message passing which typically is hidden from the application through the use of standard libraries. Operating system servers communicate with other services only through message passing, and in Factored Operating Systems, a single service which has been parallelized as a fleet of servers also only internally communicates via message passing.

Like previous microkernel operating systems [59, 46], fos uses message passing to communicate between different OS services. But, in contrast to many previous microkernel systems, fos parallelizes inside of a single service being provided. This is done by having each member of a fos service fleet run in a separate user process and not simply a thread. Also, fos focuses on how to fine-grain parallelize not only high-level services, but also low-level system services while only relying on message passing for server to server communication. In order to do this, fos leverages ideas such as lazy information update, heavy use of caching, and multi-phase commit protocols from distributed systems and distributed operating systems. Unlike distributed operating systems and Internet scale distributed systems, fos applies these concepts to low-level OS management tasks such as parallelizing memory management, process manage-

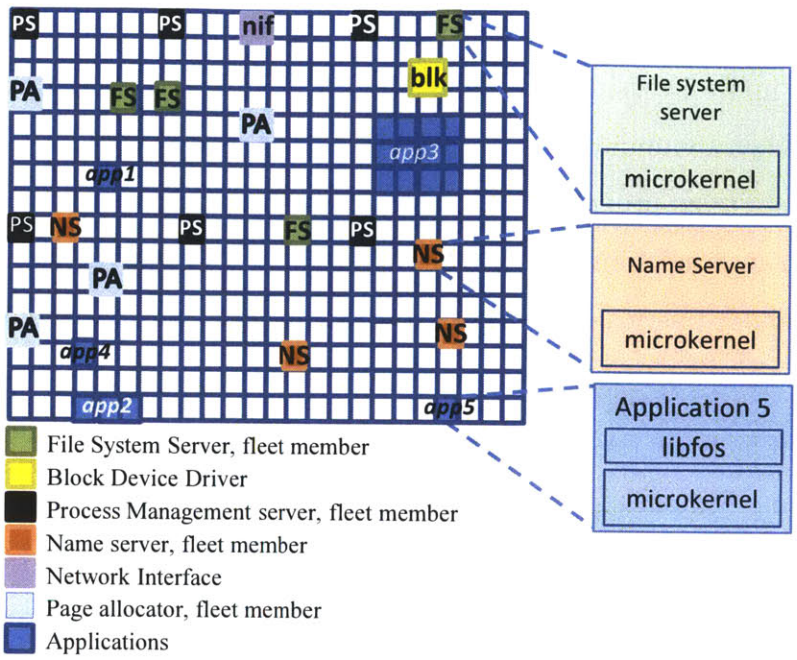

Figure 1-1: A high-level illustration of fos servers laid out across a multicore machine. This figure demonstrates that each OS service consists of several servers which are assigned to different cores. Each box in the figure represents a processor core in a multicore processor.

ment, and scheduling. Also, unlike previous systems, the scale of systems and cost of communication is quite different than distributed operating systems of the past.

fos is an experimental prototype operating system which has been constructed at MIT to explore the ideas of Factored Operating Systems. It includes a microkernel which provides messaging primitives and a protected interface to processor state. fos contains a naming system which enables the messaging system to be load balanced and for message receivers to be relocated. fos applications utilize a library called `libfos` to translate legacy system calls to messages targeting fos system servers. And finally, fos contains fleets of system servers executing in userspace which implement the core functionality of an operating system. Figure 1-1 shows a variety of applications and fos system services executing on a single multicore processor.

This thesis utilizes Factored Operating Systems and fos as the setting and context for the work. Because fos is new, we spend a portion of this thesis developing the techniques and designs used in fos to give the reader context for dPool.

1.2 fos Server Construction Challenge

One of the key components of Factored Operating Systems is the *fleet*, which is a set of server processes collaborating to provide a single system service. Example system services which are built using the fleet model include process management, physical memory management, networking, naming service, and file system service. Fleets are designed such that each process in a fleet is bound to a particular processor core and the fleet can dynamically grow and shrink the number of server processes providing a service in order to react to load in an elastic manner. Applications and other system services communicate with a fleet only via messaging and fleet members communicate with each other only via messages.

While the fleet model and messaging-only design has advantages such as making communication explicit, removing dependency on shared memory, removing dependency on complex lock hierarchies, and reducing OS-application working set cache aliasing, constructing fine-grain parallel low-level OS servers in such a restricted environment can be a burden on the systems programmer. The primary challenge for constructing parallel low-level OS services which only internally communicate with messaging is the management of shared state. Managing shared state is paramount as much of the purpose of an operating system *is* the management of resources and resource allocation. Unlike OSes which keep shared state in global shared memory and then simply utilize locks and critical sections to restrict access to state, in a shared-nothing environment such as fos, the system programmer needs to manage where to find a particular piece of data, how to keep that data up to date, and, in order to fulfill the scalability requirement of fos, provide scalable access to shared state as the number of servers in a fleet grows. These challenges to the OS fleet designer can be summarized more formally as:

- Partitioning of a shared resource across multiple system server processes.
- Providing concurrent scalable access to a shared resource across multiple server processes under even and uneven loads.

- Maintaining a consistent view of shared state across multiple system server processes.

In addition to these challenges, an ideal solution for managing shared state would enable the solution to be applied across multiple fos service fleets. Unfortunately, many of the previous messaging-only shared state solutions have been designed for application level programs used in high performance computing (HPC). HPC applications typically have a complete software stack at their disposal including networking, MPI implementations, preemptive multithreading, and advanced programming languages with runtime support, while fos fleets need to solve these problems in the context of low-level operating system services which cannot rely on any preexisting infrastructure as fos system servers by definition are implementing system services.

1.3 dPool

This thesis addresses the challenge of shared state in fos system service fleets outlined above in Section 1.2, for one particular data structure, an unordered collection of elements. dPool is a distributed data structure which provides an interface to an unordered collection of elements. The interface to dPool is through a function call interface which contains two primary calls, `poolAdd(...)` and `poolGet(...)`. `poolAdd(...)` adds elements to the dPool collection and `poolGet(...)` removes and returns a random element out of the dPool collection. The dPool data structure is implemented within a library designed to be used by fos system service fleets. The elements stored within a dPool instance are stored within the address spaces of system server processes which share a single dPool instance and do not rely on external servers or processes to hold state. Elements within a dPool instance can be distributed among the different server processes which use the instance. Internal to the dPool data structure, only messaging is used to communicate dPool state between the server processes which contain dPool state. Some dPool implementations utilize background idle threads which can rebalance elements between different portions of a dPool instance.

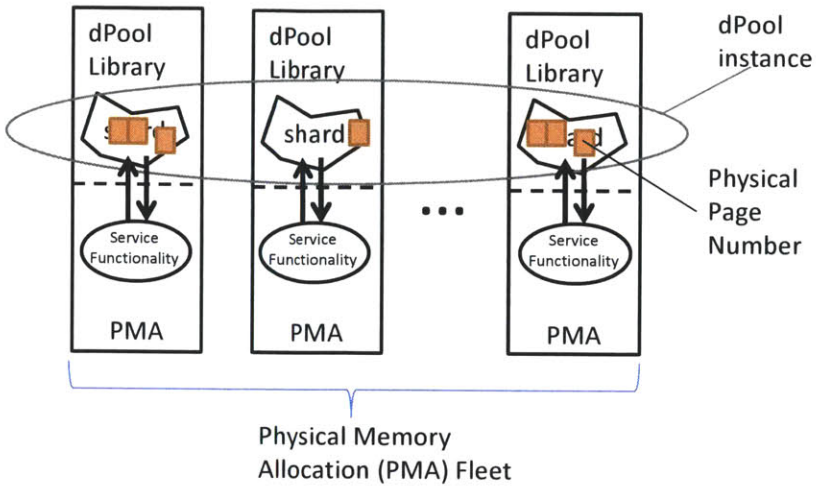

Figure 1-2: The Physical Memory Allocation Fleet using a dPool instance to manage the physical memory free page list. Each Physical Memory Allocation server process has been split into the dPool library and the service functionality. Different portions of the dPool communicate with each other via messages. Each Physical Memory Allocation server process links in the dPool library.

This thesis shows the dPool being used for the allocation of physical pages in the fos Physical Memory Allocation Server fleet and in the allocation of process identifiers from a fixed size pool for the Process Management Server fleet. The performance of dPool is characterized and exhibits good scalability for both even and uneven micro-benchmark workloads when used inside of the Physical Memory Allocation fleet. Figure 1-2 shows the dPool being used by the Physical Memory Allocation Server fleet. This thesis examines the application of parallel and distributed programming techniques as applied to the dPool library. Different algorithms being used inside dPool are evaluated and the suitability of adding background threads inside of dPool is measured.

One important consideration in the design of the dPool library is that it must fit within the fos infrastructure. One of the key aspects to fitting well with fos fleets is that the use of dPool not impose programming model requirements on the fleet designer. To enable use in a wide range of fleet design models, dPool does not require preemptive threading and can be used by sequential, user-level threaded, and preemptively threaded fos fleets. Also, because fos fleets elastically change size,

a dPool instance keeps its internal state consistent in the face of the growing or shrinking the of number of server processes utilizing it. In order to enable shrinking, the fos server process which is leaving the fleet deconstructs the local dPool. The local dPool in turn makes sure that any remaining local state is pushed to portions of the dPool in servers which will continue to run after the fleet resizes.

Because dPool is utilized by low-level OS services, the dPool implementation cannot rely on high-level primitives such as preemptive multithreading, high-level languages, advanced messaging libraries like MPI, and sophisticated memory management. To ease the development of dPool, a remote procedure call (RPC) library was constructed, along with a user-level cooperative threading model and dispatch library.

1.4 Thesis Contributions

This thesis makes the following contributions:

- Details the design choices and implementation of a prototype Factored Operating System, fos.
- Shows that a message passing based, physical page allocator OS service can be split into two parts:
  - dPool
  - Main service functionality
- Provides the first implementation and detailed description of the dPool data structure.
- Describes two fos service fleets using the dPool distributed data structure.
- Shows that parallel and distributed programming techniques can be applied to the creation of dPool.

- Shows that the use of background idle threads can be used to improve the performance and scalability of dPool.
- Demonstrates that intelligently pushing elements from portions of a dPool which contain more elements to those that contain fewer provides better performance than pulling elements between dPool parts.

1.5 Outline

The rest of this thesis is organized as follows. Chapter 2 presents related work in this area, focusing on related architectures which motivate this work, previous operating systems, and finally other distributed data structures and objects. Chapter 3 describes the fos system. We spend significant time on the development of fos because in order to understand the context of dPool, the fos system needs to be well understood. Also, this thesis serves as the canonical design reference for much of the fos work, and much of what is presented in Chapter 3 is otherwise unpublished. Last, much effort and work of this thesis focused on creating the fos system. Chapter 4 describes the construction of dPool, the infrastructure needed to build dPool, and the algorithms used by dPool. Chapter 5 describes how the dPool has been integrated within two fos server fleets. In Chapter 6 we describe how we test the dPool, present results for different algorithms being used inside of dPool, and discuss how these algorithms compare. Finally, Chapter 7 presents conclusions, lessons learned, and future directions.

Chapter 2

Related Work

2.1 Multicore Processors

The work in this thesis is motivated by the advent of multicore and manycore processors. If trends continue, we will soon see single chips with 1000's of processor cores.

2.1.1 Trends

Some trends of current and future multicore processors have influenced the design of fos. One of those trends, as noted in Howard's Single-Chip Cloud Computer (SCC) work [39], is that the cost of on-chip cache coherence is expensive and hence the SCC elected not to use it. A similar insight can be seen in the Raw [72, 68], TILE64 [74], and IBM Cell [36] designs. The Raw processor is only coherent at the memory controllers, the original TILE64 design has modest performance on-chip coherence in order to save area and complexity, and the Cell processor does not have coherent caches and uses explicit DMA to transfer data between cores.

One of the major questions for future multicore chip and OS designs is whether on-chip cache coherence can be made to have suitable performance for future large scale single chip manycore processors. A detailed discussion of coherence protocols is beyond the scope of this thesis, but there are two main challenges. First, can

the amount of storage needed for bookkeeping coherence protocols be made not to dominate the on-chip storage? Second, can future coherence protocols provide good performance for widely shared structures such as those used in a operating system kernel? fos explores the design space where future multicore chips have non-scalable shared memory or do not have global coherent shared memory. By removing the requirement of coherent shared memory, fos allows computer architects to focus their design efforts on other portions of the hardware design.

A second trend that has influenced the design of fos is that some massively multicore processors are integrating on-chip message passing networks. The Raw processor, the Tilera family of processors, and the Intel SCC all have this feature in common.

2.1.2 Multicore Implementations

There have been several research projects which have designed prototypes of massively multicore processors. The MIT Raw Processor [72, 68], the Piranha Chip Multiprocessor [12], and the 80-core Intel-designed Polaris project [70] are examples of single-chip, research multiprocessors.

More recently, we have seen Intel release a research prototype Single-Chip Cloud Computer (SCC) [39] which has 48 cores. The SCC has a memory configuration where all of the cores share physical memory, but memory is not coherent until it has reached the off-chip DDR-3 memory bank. In this way, it is similar to the memory system of the Raw microprocessor which also does not support on-chip cache coherence. Also, similar to the Raw microprocessor, all of the cores are connected by a mesh network to off-chip memory. One interesting feature of the SCC is that it contains on-chip Message Passing Buffers (MPB). MPBs are essentially small distributed scratch memories that can be kept coherent on-chip through software means. They are designed to be used as a on-chip message passing primitive.

Chip multiprocessor research has begun to transition from research into commercial realization. One example is the Niagara processor [45] designed by Afara Websystems and later purchased by Sun Microsystems. The Niagara 1 processor has eight cores each with four threads and has coherent shared memory. The Niagara

2 [49] has eight cores each with eight threads and the Niagara 3 [57] has 16 cores each with eight threads. Even commercial x86 processors have begun transitioning to multicore with the release of 6-core (AMD Opteron 6000 Series) and 10-core (Intel Xeon E7 Family) true-die (all cores on a single piece of silicon in contrast to multi-chip modules which have become popular) offerings from AMD and Intel respectively.

Another commercial, massively multicore processor family includes the TILE64, TILE64Pro, and TILE-Gx processors [74] designed by Tilera Corporation. The TILE architecture utilizes a mesh topology for connecting 64 processor cores. The TILE Architecture provides register-mapped, on-chip networks to allow cores to explicitly communicate via message passing. TILE processors also support shared memory between all of the cores. The TILE Architecture supports multiple hardware levels of protection and the ability to construct hardwalls which can block communications on the on-chip networks.

The research presented in this thesis envisions that future processors will look like TILE Architecture processors scaled up to 1000's of cores. This work supposes that future processors will contain on-chip networks, many protection levels, and hardwall capabilities. The fos operating system will suppose these features integrated with the industry standard x86 instruction set.

2.2 OSes

There are several classes of systems which have similarities to fos. These can be roughly grouped into three categories: traditional microkernels, distributed operating systems, and distributed systems.

2.2.1 Microkernels

A microkernel is a minimal operating system kernel which typically provides no high-level operating system services in the kernel, but rather provides mechanisms such as low-level memory management and inter-thread communication which can be utilized to construct high-level operating system services. High-level operating system services

such as file systems and naming services are typically constructed inside of user-level servers which utilize the microkernel's provided mechanisms. Mach [2] is an example of a microkernel. In order to address performance problems, portions of servers were slowly integrated into the Mach microkernel to minimize microkernel/server context switching overhead. This led to the Mach microkernel containing more functionality than was originally envisioned. The L4 [46] kernel is another example of a microkernel which attempts to optimize away some of the inefficiencies found in Mach and focuses heavily on performance. Microkernels have also been used in commercial systems. Most notably, the QNX [37] operating system is a commercial microkernel largely used for embedded systems.

Mach Multiserver (Mach-US) [59] is a microkernel OS which warrants special attention. The Mach Multiserver project worked hard to split out the different portions of the UNIX functionality into different servers. This is in contrast to other Mach implementations which put much of the UNIX functionality into one server for performance and ease of construction. The first level of factorization of fos is very similar to how Mach-US attempted to split UNIX services into separate servers. Because fos is being designed in a processor core rich environment which did not exist when Mach-US was explored, fos goes one step further and parallelizes within system services.

fos is designed as a microkernel and extends microkernel design. fos leverages many of the lessons learned from previous microkernel designs. It is differentiated from previous microkernels in that instead of simply exploiting parallelism between servers which provide different functions, this work seeks to distribute and parallelize within a service for a single OS function. In example, the fos physical memory allocator is split into different processes that communicate via messaging. By splitting physical memory allocation into separate processes, more physical memory allocation throughput can be achieved versus having a single server process. This work also exploits the spatial nature of massively multicore processors. This is done by spatially distributing servers which provide a common function. This is in contrast to traditional microkernels which were not spatially aware, as most previous microkernels

were designed for uniprocessor or low core-count systems. By spatially distributing servers which collaboratively provide a high-level function, applications which use a given service may only need to communicate with the local server providing the function and hence can minimize intra-chip communication. Operating systems built on top of previous microkernels have not tackled the spatial non-uniformity inherent in massively multicore processors.

The cost of communication on fos compared to previous microkernels can reduced because fos does not temporally multiplex operating system servers and applications. Therefore, when an application messages a fos OS server, a context swap does not occur. This is in contrast to previous microkernels which temporally multiplexed resources, causing communication from one process on a processor to a different process on the same processor to require a context swap. Last, fos, is differentiated from previous microkernels on parallel systems, because the communication costs and sheer number of cores on a massively multicore processor is different than in previous parallel systems, thus the optimizations made and trade-offs are quite different.

The Tornado [32] operating system which has been extended into the K42 [4] operating system is one of the more aggressive attempts at constructing scalable microkernels. They are differentiated from fos in that they are designed to be run on SMP and NUMA shared memory machines instead of single-chip massively multicore machines. Tornado and K42 also suppose future architectures which support efficient hardware shared memory. fos does not require architectures to support intra-machine shared memory as communication between fos servers is all via message passing. Also, the scalability [6] of K42 has been focused on machines with up to 24 processors, which is a modest number of processors when compared to the fos design target of 1000+ processors.

The Hive [25] operating system utilizes a multicellular kernel architecture. This means that a multiprocessor is segmented into cells which each contain a set of processors. Inside of a cell, the operating system manages the resources inside of the cell like a traditional OS. Between cells, the operating system shares resources by having the different cells message and allowing safe memory reads. Hive OS focused heavily

on fault containment and less on high scalability than fos does. Also, the Hive results are for scalability up to 4 processors. In contrast to fos, Hive utilizes shared memory between cells as a way to communicate.

Another approach to building scalable operating systems is the approach taken by Disco [23] and Cellular Disco [34]. Disco and Cellular Disco run off the shelf operating systems in multiple virtual machines executing on multiprocessor systems. By dividing a multiprocessor into multiple virtual machines with fewer processors, Disco and Cellular Disco can leverage the design of pre-existing operating systems. They also leverage the level of scalability already designed into pre-existing operating systems. Disco and Cellular Disco also allow for sharing between the virtual machines in multiple ways. For instance in Cellular Disco, virtual machines can be thought of as a cluster running on a multiprocessor system. Cellular Disco utilizes cluster services like a shared network file system and network time servers to present a closer approximation of a single system image. Various techniques are used in these projects to allow for sharing between VMs. For instance memory can be shared between VMs so replicated pages can point at the same page in physical memory. Cellular Disco segments a multiprocessor into cells and allows for borrowing of resources, such as memory, between cells. Cellular Disco also provides fast communication mechanisms which break the virtual machine abstraction to allow two client operating systems to communicate faster than they could via a virtualized network-like interface. VMWare has adopted many of the ideas from Disco and Cellular Disco to improve VMWare's product offerings. One example is VMCI Sockets [71] which is an optimized, communication API which provides fast communication between VMs executing on the same machine.

Disco and Cellular Disco utilize hierarchical, shared information to attack the scalability problem much in the same way that fos does. They do so by leveraging conventional SMP operating systems at the base of the hierarchy. Disco and Cellular Disco argue leveraging traditional operating systems as an advantage, but this approach likely does not reach the highest level of scalability as a purpose built scalable OS such as fos. For example, the rigid cell boundaries of Cellular Disco can limit scal-

ability. Because these systems are just utilizing multiprocessor systems as a cluster, the qualitative interface of a cluster is restrictive when compared to a single system image. This is especially prominent with large applications which need to be rewritten such that the application is segmented into blocks only as large as the largest virtual machine. In order to create larger systems, an application needs to either be transformed to a distributed network model, or utilize a VM abstraction-layer violating interface which allows memory to be shared between VMs. Also, in contrast to fos, Disco and Cellular Disco focus on parallelizing high level system services such as the file system, while fos also focuses on fine-grain parallelization of low-level system services.

2.2.2 Distributed Operating Systems

fos bears much similarity to a distributed operating system, except executing on a single multicore chip. In fact, much of the inspiration for this work comes from the ideas developed for distributed operating systems. A distributed operating system is an operating system which executes across multiple computers or workstations connected by a network. Distributed operating systems provide abstractions which allow a single user to utilize resources across multiple networked computers or workstations. The level of integration varies with some distributed operating systems providing a single system image to the user, while others provide only shared process scheduling or a shared file system. Examples of distributed operating systems include the V Distributed System [27], Amoeba [67, 66], Sprite [50], Choices [24], and Clouds [31]. These systems were implemented across clusters of workstation computers connected by networking hardware. One important aspect of distributed OSes that fos leverages is that both applications communicate with servers and inter-machine OS servers communicate with each other via messaging.

While fos takes much inspiration from distributed operating systems, some differences stand out. The prime difference is that the core-to-core communication cost on a single-chip, massively multicore processor is orders of magnitude smaller than on distributed systems which utilize Ethernet style hardware to interconnect the nodes.

Single-chip, massively multicore processors have much smaller core-to-core latency and much higher core-to-core communications bandwidth. fos takes advantage of this by allowing finer-grain paralelization of system services which typically requires more communication than coarse-grain parallelization. A second difference that multicores present relative to clusters of workstations is that on-chip communication is more reliable than workstation-to-workstation communication over commodity network hardware. fos takes advantage of this by approximating on-chip communication as being reliable. This removes the latency and complexity of correcting communication errors.

Single-chip, multicore processors are easier to think of as a single, trusted administrative domain than a true distributed system. In many distributed operating systems, much effort is spent determining whether communications are trusted. This problem does not disappear in a single-chip multicore, but the on-chip protection hardware and the fact that the entire system is contained in a single chip simplifies the trust model considerably.

Also, in contrast to many previous distributed operating systems, fos fine-grain parallelizes lower level services while previous distributed OSes have focused on coarser parallelization and higher level system services. For example, previous distributed OSes have focused on parallelizing high-level services such as file servers. fos takes this level of fine-grain parallelization one step further by using fine-grain parallelization techniques on low-level services such as process management, scheduling, and memory management. Previous systems such as Amoeba have parallelized resources such as process management, but the parallelization was done on a per user basis which is much coarser than the level of parallelization that fos uses. Also, in contrast to fos, Ameoba utilized threading and shared memory for communication for parallel system servers when executing on a single computer. fos instead utilizes messaging for intra-fleet communication.

More recently, work has been done to investigate operating systems for multicore processors. One example is Corey [20] which focuses on allowing applications to direct how shared memory data is shared between cores. Corey also investigates

exploiting the spatial nature of multicore processors by dedicating cores to portions of the application and operating system. This is similar to how fos dedicates cores to particular OS functions.

A contemporary of the fos project which is tackling many of the same challenges is the Barrelfish [13] Operating System. Barrelfish is a based around a multikernel design. Barrelfish defines a multikernel as an operating system kernel which treats a multiprocessor as a network of independent cores. It moves traditional OS functionality into servers executing as user-level servers. In Barrelfish, each core has what it terms the monitor process and these processes use state replication and two phase commit protocols to keep OS state coherent. One difference between fos and the Barrelfish design is that Barrelfish puts much of the OS functionality into the monitor process while fos factors OS functionality into service specific fleets which do not execute on the same core as the application. As per recent discussions with the Barrelfish group, they are moving more functionality out of the monitor process and factoring the OS more by function much in the same way that fos factors by service provided. There are two areas that Barrelfish has excelled when compared to fos. First, the Barrelfish project has spent more effort optimizing their messaging system for manycore systems. Second, Barrelfish has explored using a database [53] to make intelligent decisions about optimizing for different multicore systems where the diversity and nonuniformities of the system effect the OS greatly. fos is further along in exploring parallelizing different servers and has also been extended across clusters and clouds which Barrelfish has yet to be extended.

2.2.3 Distributed Systems

The manner in which fos parallelizes system services into fleets of cooperating servers is inspired by distributed Internet services. For instance, load balancing is one technique fos leveraged from clustered webservers. The name server of fos derives inspiration from the hierarchical caching in the Internet's DNS system. fos hopes to leverage other techniques such as those in peer-to-peer and distributed hash tables such as Bit Torrent [29] and Chord [60]. fos also takes inspiration from distributed

services such as distributed file systems like AFS [52], OceanStore [42] and the Google File System [33].

While this work leverages techniques which allow distributed Internet servers to be spatially distributed and provide services at large-scale, there are some differences. First, instead of being applied to serving webpages or otherwise user services, these techniques are applied to services which are internal to an OS kernel. Many of these services have lower latency requirements than are found on the Internet. Second, the on-chip domain is more reliable than the Internet, therefore there is less overhead required to deal with errors or network failures. Last, the communication costs within a chip are orders of magnitude lower than on the Internet.

2.3 Distributed Data Structures

The dPool data structure has much in common with and takes inspiration from previous parallel object models and parallel container classes. In this section, we will compare and contrast previous parallel object models with fos's dPool. One of the main differences between dPool and previous parallel container classes is that dPool is designed to be used inside of low-level OS services. This has a large impact on the design of dPool. First, dPool cannot rely on high level constructs such as MPI [35], preemptive threads, and complex schedulers because dPool itself can be used to implement some of these low-level features. Also, dPool does not utilize an object-oriented language because it has been designed to be used by fos fleet servers which are typically written in straight 'C'. Most of the related parallel objects have been designed for HPC environments running on a cluster which is quite a different environment than being used inside of an OS. There have been several projects which have looked at using parallel objects inside of operating systems, but many of them utilize shared memory to communicate. Finally, some of the high performance computing data structures have not been designed around elastic resizing. The lack of resizing in a HPC environment is largely because machine size is traditionally fixed and assigning data to machines statically simplifies the design.

2.3.1 Data Structures Designed for Operating Systems

The first related project which we will examine is Clustered Objects [5], which is used inside of the K42 operating system. Clustered Objects provides a common infrastructure on which to build distributed objects to be used in the K42 operating system. The Clustered Object infrastructure is based off of 'C++' and utilizes sophisticated virtual pointer table manipulation in order to allow different processors to invoke a CPU-specific *Representative* given the same reference. Representatives are similar to dPool's *shard*s. Clustered Objects allows a local Representative to be the interface to shared data stored in a Clustered Object instance. Clustered Objects provides a programming model by which a common interface can be used to access a Clustered Object while local Representatives can be used to implement the functionality and sharing of data for the object internally. Although Clustered Objects is an interface and set of libraries that facilitate writing distributed shared data structures, it is not a complete set of parallel classes. During the implementation of K42, a number of Clustered Objects have been written with varying degrees of distribution.

Much like dPool, Clustered Objects was designed to be used to implement low-level, shared data structures inside of operating systems. One of the primary differences between Clustered Objects and the fos approach is that Clustered Objects internally use shared memory to communicate. The different Representatives contain pointers to a Root structure where they can gain access to other Representatives. Clustered Objects does not restrict and in fact encourages utilizing shared memory data when it makes sense. Internally, Clustered Objects do not use message passing as the K42 project was focused on running on shared memory NUMA machines. In contrast, dPool *shard*s are in different address spaces and can only communicate via explicit message passing. Another difference between Clustered Objects and dPool is that dPool has been designed to be a container class which can store any data type. While it is possible to build such a Clustered Object, the philosophy of Clustered Objects was to encapsulate more functionality inside of a single Clustered Object than to implement simple container classes. Instead, Clustered Objects typically were ap-

plication specific and contained service functionality along with the data structure. In fos, we put the service functionality in the fleet member and let the data structure be simply a container. Also, Clustered Objects do not allow background threads to execute inside of the Clustered Objects while dPool does. dPool's background threads can be used to balance elements within the dPool data structure off of the critical path of the computation. Background threads harvest what would otherwise be idle cycles on the processor. Last, I was not able to find any reference to a Clustered Object which provided an unordered set (pool) style data structure having been implemented as a Clustered Object.

Fragmented Objects [47, 55, 22] is a distributed object model which puts fragments of an object in the address space of the application using the object and allows the fragmented object writer to hide communications between fragments. The fragments communicate via messaging. Fragmented Objects bears much resemblance to the design of dPool. Fragmented Objects were designed in the context of distributed, multi-machine, systems and used 'C++' to implement the objects. Fragmented Objects do not enable background threads to execute inside of the Fragmented Objects, in contrast to dPool. The Fragmented Objects work focused more on the properties of such objects than in implementations built using the methodology.

In the SOS project [56], Fragmented Objects were used inside of the SOS operating system. SOS is an object-oriented operating system which ran on top of UNIX (SunOS) as a meta-OS. SOS also had a multi-machine, distributed system mode. SOS used UNIX Domain Sockets for the intra-machine communication and IP for the inter-machine case to allow the different portions of the OS to communicate. This is similar to how fos uses its messaging layer to communicate between different fleet servers. The SOS system was more of a meta-OS than a true OS, as it deferred handling memory management and scheduling to the host OS, SunOS. The SOS project was more focused on how to construct object-oriented systems than how to optimize OS services. As such, they did not focus on the parallelization of the Fragmented Objects used inside. In fact, all of the examples given show centralized implementations of objects where the local fragment of the object was simply a Remote Procedure

Call (RPC) proxy to the centralized implementation of the object. Another key difference between dPool and SOS's use of Fragmented Objects is that SOS was focused on uniprocessor and clusters of uniprocessor systems versus fos's focus on multicore systems.

Distributed Shared Objects [65, 9] and later extensions [69, 38] were object models developed at Vrije Universiteit for use in both the construction of OSes and applications. They were used in concert with the Orca [8] programming language and the Ameoba distributed operating system. In distributed shared objects, different portions of a shared object communicate via messaging. The structure of dPool is similar to shared objects in that they both replicate and partition state in the local address space of users of the distributed objects. In contrast to dPool, Orca and shared objects were focused on implementations on multi-machine clusters while dPool focuses on implementing scalable data structures on a multicore processor. One other difference is that most of the implementations of Orca were based on distributed shared memory to keep state coherent in a general manner while dPool encapsulates the sharing and partitioning of data within the data structures themselves. This flexibility allows dPool to have background threads within the data structure implementations to rebalance the elements held within the data structure.

2.3.2 Data Structures Designed for Applications

The Standard Template Adaptive Parallel Library(STAPL) [63, 62] is a parallel data structure library which provides a set of scalable, concurrently accessible, container classes with functionality similar to the C++ Standard Template Library(STL). STAPL is written on top of the ARMI communication library which enables different portions of a `pContainer` to communicate either via shared memory or via a message passing interface. STAPL is largely designed to be used by large scale application writers and is hence optimized for the High Performance Computing (HPC) community. One very interesting feature of STAPL containers is that they are composable. This allows one `pContainer` to be stored inside of another `pContainer`. STAPL data structures are adaptable in the manner in which they place and partition elements.

Some STAPL implementations allow the spatial mapping of stored elements to be changed in response to load. This is similar to how dPool is able to use background threads to rebalance elements. Because STAPL objects are composable, this limits optimizations that dPool implementations are able to use.

Unlike dPool, STAPL `pContainers` are written requiring 'C++' allows STAPL to take advantage of templates while dPool must use a more dynamic interface for the size of elements that dPool stores. STAPL data structures are also designed for use by HPC applications and as such STAPL relies on a feature-rich set of underlying libraries such as MPI, Pthreads, and a complete standard 'C' library which are not present inside of an OS kernel. The STAPL heavyweight runtime system contains a communication library, a scheduler, an executor, and a performance monitor. STAPL provides a feature-rich set of container classes, and even provides a multiset which is similar to the dPool but with an expanded interface. The STAPL `pMultiSet` [64] still imposes order in the same way that STL MultiSets impose ordering. STAPL is under continued development and portions of it are contemporary with the fos effort.

In addition to STAPL, there are other parallel container libraries and data structures for applications which utilize only messaging to communicate internally. Examples include Topologies [54] and Distributed Shared Abstractions [28] which were designed to map data structures across structured MIMD machines.

There are many other parallel container libraries which rely solely on shared memory. Examples of these include Intel's Threaded Building Blocks (TBB) [40], and POOMA [51]. These libraries have all been designed to supporting application level programming and not OS programming.

Parallel programming languages such as CHARM++ [41], X10 [26], SplitC [30], and Titanium [76] typically include parallel arrays and include language primitives which ease the development of parallel data structures. The design of dPool has taken the conservative approach of providing a 'C' interface and not utilizing a programming language designed for parallelism. This primarily has been done because the language and runtime associated with many of these parallel languages is not appropriate for use in the context of operating system programming.

2.4 Multisets and Bags

The dPool data structure developed in this thesis implements the interface of a multiset which is sometimes also known as a bag data structure. Kuchen and Gladitz provide an overview of parallel bags which have been used inside of functional languages [43] and the GAMMA programming language is a functional programming language where multisets are the primary data structure used for all computation [10].

Afek [3] examines a parallel pool data structure being used for producer-consumer applications. Sundell [61] have recently been investigating concurrent bag data structures for multicore systems which do not use locks. This work still relies on shared memory through the use of compare-and-swap instructions unlike the dPool which only utilizes messaging.

Leiserson and Schardl [44] formalize the bag interface with additional operations which allow the parallel union and splitting of a bag. They use a bag to implement a parallel breadth-first search and their experimental results use a shared memory Cilk model.

2.5 Work Piles and Queues

Work piles are many times implemented with a bag or pool data structure. One interesting work pile is the parallel pool of ready to be executed instruction sequences in a dataflow computer architecture [7]. Unlike other programming models, these threads are independent. Typical dataflow architectures implemented work sharing and queues to load balance threads across the execution units.

The Cilk programming environment utilizes a parallel work queue for thread scheduling [19]. In Blumofe and Leiserson's work, they propose utilizing work stealing instead of a work sharing approach. These approaches keep independent queues per processor core and either steal work from another queue when a processor's queue is empty, or in the case of work sharing, will put work on queues of other processors when new threads are created. In contrast to a generic pool data structure such as

what dPool implements, the Cilk scheduler maintains priorities between the different threads in the system as there are order dependencies between the threads in the scheduler. One interesting result out of the Cilk work is that they found that a work stealing scheduler is superior to a work sharing scheduler. This is in contrast to what we find for dPool; that pushing elements between different portions of a dPool, similar to work sharing, is superior to pulling, similar to work stealing. The contrary findings may be because the Cilk scheduler has a global ordering that needs to be maintained, the Cilk scheduler utilizes shared memory to manage the queue, and the Cilk analysis takes in to account the running time of different threads. The contrary findings suggests that dPool with a push implementation may not be a good fit for a Cilk style thread scheduler.

Chapter 3

Structure of fos

Much of the work of this thesis has gone into the identification of the challenges in constructing operating systems for future architectures, the development of the structure of a Factored Operating System (fos), and the implementation of a prototype fos operating system framework. This chapter describes the fos system which is the context within which dPool has been created.

3.1 Motivation

3.1.1 Architecture

Future multicore chip architecture has motivated the design of fos. In the future, I believe that we will continue to see increasing numbers of cores on a single piece of silicon. This is largely being driven by the progression of Moore's Law supplying chip designers more silicon real estate. Some challenges to this vision are that future chips may become power limited at some point along this design path or circuit integration technology may put Moore's Law in peril. fos is designed around the challenge of using large numbers of cores both for the operating system and allowing applications to use large numbers of cores. The key goal for the design of fos is to allow a system to run a large number of applications on a future multicore processor and provide access to OS system services while not having the OS become the scalability limiter. Therefore

much of fos's design has been motivated by having scalability as the primary design constraint.

Another important architectural trend which has influenced the design of fos is the rise of on-chip direct communication networks and the unknown scaling of cache coherent shared memory on future multicore processors. Section 2.1 has more discussion of processor architecture. The rise of on-chip direct networks and the uncertainty of shared memory has motivated fos to use explicit communication in the form of message passing. In fos, we take the extreme view that global cache coherent shared memory will either be unavailable on future architectures or its performance will be very low and ultimately limit the scalability of any application or OS which utilizes it widely. To support this extreme viewpoint, fos only allows applications to connect to system services via message passing. System servers only communicate with each other via message passing. Services are internally parallelized, but internally communicate via messaging. In effect, fos holds the OS programmer to a high standard of explicitly thinking about all communication in the hope of leading to ultimately higher scalability.

3.1.2 Challenges of Scaling Monolithic OSes

This sub-section investigates three main scalability problems with contemporary monolithic OS design: locks, locality aliasing, and reliance on shared memory. Case studies are utilized to illustrate how these problems appear in a contemporary OS, Linux, on modern multicore x86_64 hardware.

Shared Memory Locks

Contemporary operating systems which execute on multiprocessor systems have evolved from uni-processor operating systems. The most simplistic form of this evolution was the addition of a single big kernel lock which prevents multiple threads from simultaneously entering the kernel. Allowing only one thread to execute in the kernel at a time greatly simplifies the extension of a uni-processor operating system to mul-

tiple processors. By allowing only one thread in the kernel at a time, the invariant that all kernel data structures will be accessed by only one thread is maintained. Unfortunately, one large kernel lock, by definition, limits the concurrency achievable within an OS kernel and hence the scalability. The traditional manner to further scale operating system performance has been to successively create finer-grain locks thus reducing the probability that more than one thread is concurrently accessing locked data. This method attempts to increase the concurrency available in the kernel.

Adding locks into an operating system is time consuming and error prone. Adding locks can be error prone for several reasons. First, when trying to implement a fine grain lock where coarse grain locking previously existed, it is common to forget that a piece of data needs to be protected by a lock. Many times this is caused by simply not understanding the relationships between data and locks, as most programming languages, especially those commonly used to write operating systems, do not have a formal way to express lock and protected data relationships.

The second manner in which locks are error prone is that locks can introduce circular dependencies and hence cause deadlocks to occur. Many operating systems introduce lock acquisition hierarchies to guarantee that a circular lock dependence can never occur, but this introduces significant complexity for the OS programmer. An unfortunate downside of lock-induced deadlocks is that they can occur in very rare circumstances which can be difficult to exercise in normal testing.

When the lock granularity finally needs to be adjusted, it is usually not the case that simply adjusting the lock granularity is enough. For code which has already been parallelized, it is typically difficult to make code finer grain locked in a vacuum. Instead, it is typical for entire sub-systems of the operating system to be redesigned when lock granularity needs to be adjusted.

In previous multiprocessor systems, the speed at which parallelism increased was slow and sub-system redesign was feasible. In sharp contrast, future multicore processors will follow an exponential growth rate in the number of cores. The effect of this is that each new generation of chip will require the granularity of a lock to be halved in order to maintain performance parity. Thus, this lock granularity change may re-

quire operating system sub-systems to be redesigned with each new chip generation. Unfortunately for the operating system programmer, it is very difficult to redesign sub-systems with this speed, as programmer productivity is not scaling with number of transistors. Hence, we believe that traditional, lock-based operating systems need to be rethought in light of the multicore era.

Whenever discussing lock granularity, the question arises, what is the correct lock granularity? If lock granularity is chosen to be too coarse, the scalability on highly parallel systems may be poor. But, if the lock granularity is too fine, the overhead of locking and unlocking too often can cause inefficiencies on low core-count systems. Even if a lock is not being contended, extra atomic operations are utilized to lock and unlock the memory location associated with a lock when compared to not having a lock at all. Future operating systems will have to directly attack finding the correct lock granularity as they will have to span multiple generations of computer chips which will vary by at least an order of magnitude with respect to core count. Also, the difference in core count between the high end processor and low end processor of the same generation may be at least an order of magnitude in the 1000+ core era. Thus, even within a processor family, the OS designer may not be able to choose an appropriate lock granularity.

Case Study: Physical Page Allocator In order to investigate how locks scale in a contemporary operating system, I investigated the scaling aspects of the physical page allocation routines of Linux. The Linux 2.6.24.7 kernel was utilized on a 16 core Intel quad-socket quad-core system. The test system is a Dell PowerEdge R900 outfitted with four Intel Xeon E7340 CPUs running at 2.40GHz and 16GB of RAM.

The test program attempts to allocate memory as quickly as possible on each core. This is accomplished by allocating a gigabyte of data and then writing to the first byte of every page as quickly as possible. By touching the first byte in every page, the operating system is forced to demand allocate the memory. The number of cores was varied from 1 to 16 cores. Precision timers and `oprofile` were utilized to determine the runtime and to profile the executing code. Figure 3-1 shows the results of this

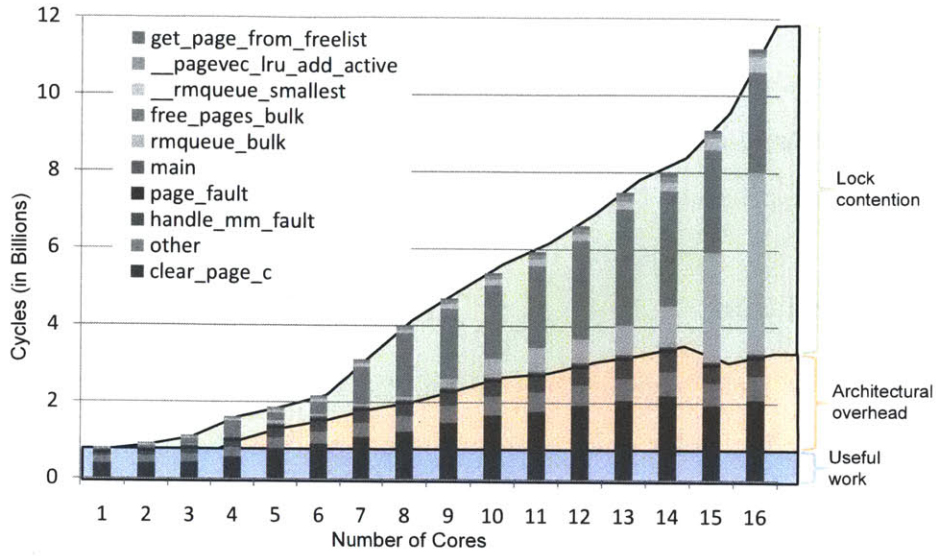

Figure 3-1: Physical memory allocation performance sorted by function. As more cores are added, more processing time is spent contending for locks.

experiment. The bars show the time taken to complete the test per core. Note that a fixed amount of work is done per core, thus perfect scaling would be bars all the same height.

By inspecting the graph, several lessons can be learned. First, as the number of cores increases, the lock contention begins to dominate the execution time. Beyond eight processors, the addition of more processors actually slows down the computation and the system begins to exhibit fold-back. We highlight architectural overhead as time taken due to the hardware not scaling as more cores are added. The architectural overhead is believed to be caused by contention in the hardware memory system.

For this benchmark, the Linux kernel already utilizes relatively fine-grain locks. Each core has list of free pages and a per-core lock on that free list. When the local list of free pages becomes dry, the page allocation code locks a global, per-NUMA node list and moves pages from that free list to the per-CPU cache. If the local per-CPU cache gains too many pages, it pushes pages back to the global, per-NUMA node free list. The global, per-NUMA node free list is kept as a buddy allocator.

Even with all of these optimizations, the top level per-NUMA node re-balancing lock ends up being the scalability problem. This code is already quite fine-grain locked, thus, to make it finer grain locked, some algorithmic rethinking is needed.

While it is not realistic for all of the cores in a 16 core system to allocate memory as quickly as this test program does, it is realistic that in a 1000+ core system, 16 out of the 1000 cores would need to allocate a page at the same time thus causing traffic similar to this test program.

Working Set Aliasing

Operating systems have large instruction and data working sets. Traditional operating systems time multiplex computation resources. By executing operating system code and application code on the same physical core, implicitly shared resources such as caches and TLBs have to accommodate the shared working set of both the application and the operating system code and data. This can reduce the hit rate in these cache structures versus executing the operating system and application on separate cores. By reducing cache hit rates, the single stream performance of the program will be reduced. Reduced hit rate is exacerbated by the fact that manycore architectures typically contain smaller per-core caches than past uniprocessors. If the OS and application are communicating often, for instance when passing large portions of data between the OS and application, positive cache interference can occur also.

Single-stream performance is at a premium with the advent of multicore processors, as increasing single stream performance by other means may be exceedingly difficult. It is also likely that some of the working set will be so disjoint that the application and operating system can fight for resources, causing anti-locality collisions in the cache. Anti-locality cache collisions are when two different sets of instructions pull data into the cache at the same index hence causing the different instruction streams to destroy temporal locality for data at a conflicting index in the cache. Current operating systems also execute different portions of the OS with wildly different code and data on one physical core. By doing this, intra-OS cache thrash can be accentuated versus when executing different logical portions of the OS on different physical cores.

Cache interference also hampers embedded operating systems which offer quality of service (QOS) or real-time guarantees. The variability introduced by OS-

application cache interference has caused many embedded applications to eliminate usage of an operating system and elect to use a more bare-metal approach.

Case Study: Cache Interference

In order to evaluate the cache system performance degradation due to executing the operating system and application code on the same core, I created a cache tool which allows us to differentiate operating system from application memory references. The tool is based off of the x86_64 version of QEMU, and captures full system memory references differentiated by protection level. Adam Belay and I extended this tool into a tool based on CoreEMU which we call CacheEMU [16] which can determine the cache miss rates attributable to the operating system, the application, and interference or cooperation misses caused by the operating system and application contending for cache space. This was accomplished by simulating a unified cache, an OS only cache, and an application only cache for differing cache sizes and configurations.

We conducted a study using CacheEMU to simulate a 64-bit x86 computer executing Debian Linux. We chose five common Linux workloads with heavy usage of OS services. They are as follows:

- **Apache:** The Apache Web Server, running Apache Bench tests 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 [21], indexing the entire Linux Kernel source tree.
- **Zip:** The standard compressed archive tool, packing the entire Linux Kernel source tree into a zip archive.

In the following experiments, an 8-way set associative cache was simulated and the cache size was varied from 4KB to 16MB. In each test, we compared the number

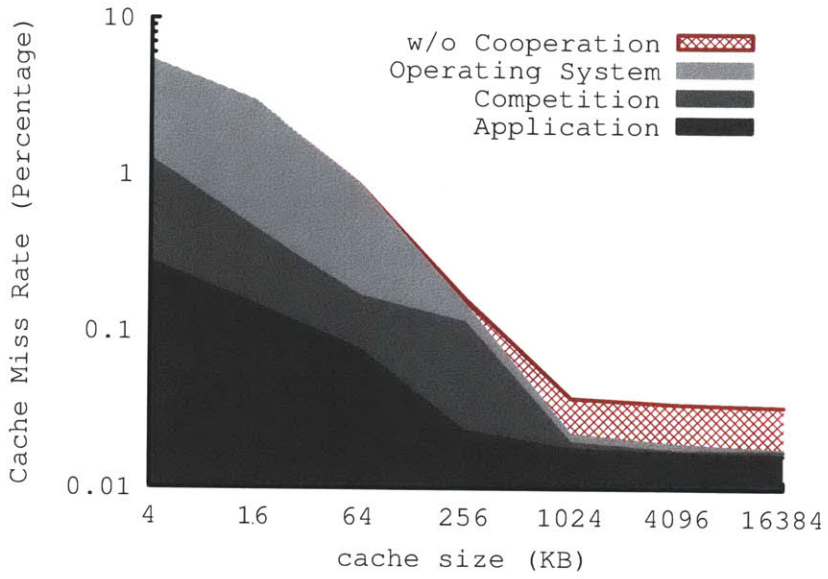

Figure 3-2: Cache miss rates for Zip running on Linux vs Cache size. Shows misses attributable to Application, OS, OS-Application conflict/competition, and misses that go away due to cache cooperation.

of misses when utilizing a separate OS and application cache versus a cache that is used by both the application and OS. In general we found that sharing a cache for small sized caches caused significant cache competition, while cooperation became a significant factor for larger cache sizes.

Figure 3-2 shows the cache behavior for gzip executing on Linux. Studying these results, it can be seen that for small cache sizes, the miss rates for the operating system far surpass the miss rates for the application. Second, for small cache sizes, the miss rate due to cache interference is sizable. We found that for most of our benchmarks, misses due to the OS overwhelm that of the application. Figure **??** shows that the misses caused by the OS overwhelm the misses caused by the application for all cache sizes, but this may be difficult to see as this figure shows the data plotted on a log plot. Cache competition/interference was dominant until the cache size reached 1MB and then cache cooperation between the OS and application took over.

Figure 3-3 shows results for all five test applications. This graph shows the per-

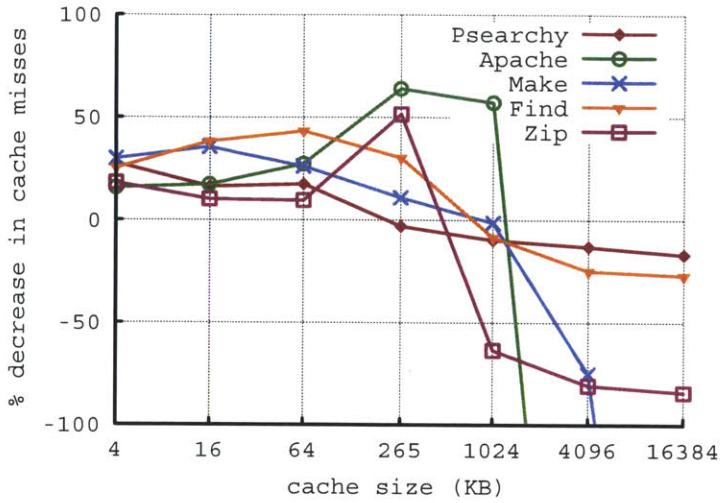

Figure 3-3: Percentage decrease in cache misses caused by separating the application and OS into separate caches of the same size versus cache size. Negative values denotes that performance would be better if sharing a cache.

centage that cache misses decrease when the OS and application use separate caches of the same size. This has been plotted against cache size. Note that the case where the OS and application is split apart has twice as much cache as the case when the OS and application share caches. When the percent decrease is above zero, it indicates that the application and OS would run faster if split onto separate cores. When negative, it suggests that the OS and application benefit from being scheduled on the same core. As Figure 3-3 shows, for small cache sizes (4KB-256KB), those typically found in L1 and L2 caches, benefit can be had by not having the OS and application share caches. For larger sized caches (1MB-16MB), performance would be increased by sharing a cache of that size. These results indicate that it would make sense to execute the OS and application on different cores, but share a last level or L3 cache between the application and OS that services it. Further discussion of this topic can be found in Belay [16]. The FlexSC [58] work took some inspiration from my original studies in the fos proposal paper [73] and develops a similar idea of moving the OS and application onto separate cores in the context of a traditional Linux system.

Reliance on Shared Memory

Contemporary, monolithic operating systems rely on shared memory for communication. Largely, this is because shared memory is the only means by which a desktop hardware architecture allows core-to-core communication. The abstraction of a flat, global, address space is convenient for the programmer to utilize as addresses can be passed across the machine and any core is capable of accessing the data. It is also relatively easy to extend a single threaded operating system into a multi-threaded kernel by using a single global address space. Unfortunately, the usage of a single global shared memory is an inherently global construct. This global abstraction may make it challenging for a shared memory operating system to scale to large core count if the hardware does not support efficient global shared memory.

Many current embedded multicore processors do not support a shared memory abstraction. Instead cores are connected by ad-hoc communication FIFOs, explicit communication networks, or by asymmetric shared memory. Current day embedded multicores are pioneers in the multicore field which future, general-purpose multicore processors will extend. Because contemporary operating systems rely on shared memory for communication, it is not possible to execute them on current and future embedded multicores which lack full shared memory support. In order to have the widest applicability, future multicore operating systems should not be reliant on a shared memory abstraction.

It is also unclear whether cache coherent shared memory will scale to large core counts. Although the most promising hardware shared memory technique with respect to scalability has been directory based cache coherence, hardware directory based cache coherence has found difficulties providing high-performance, cache coherent shared memory above about 100 cores. The alternative is to use message passing, which is a more explicit point-to-point communication mechanism.

Besides scalability problems, modern operating system's reliance on shared memory can cause subtle data races. If used incorrectly, global shared memory easily allows the introduction of data races which can be difficult to detect at test time.

3.1.3 fos's Response to Scalability Challenges

In the above sections, we identified these three challenges of scaling monolithic operating systems to future multicore processors:

- Shared Memory Locks
- OS-Application Working Set Aliasing
- Reliance on Shared Memory

fos addresses each of these scalability limiters. To address the problems in finding the correct lock granularity and the composability challenges of complex lock hierarchies, fos does not use shared memory locks inside of system servers.

To address working set conflicts of executing the OS and application or different portions of an OS on the same core, fos dedicates cores to OS system servers and applications. Also, fos factors the OS by operating system function, therefore different portions of the OS which have non-overlapping working sets also execute on different cores. Applications communicate with OS servers via message passing. Also OS servers communicate with other OS servers also via message passing.

Finally fos breaks the OS's reliance on shared memory. fos achieves this by only internally using message passing to communicate between different OS system servers. fos does support applications that use shared memory as a way to broaden the range of applications that fos can execute, but internally fos system servers do not use shared memory.

3.2 fos Design

fos has been designed to allow an operating system to scale up and use large numbers of cores in a multicore system. In order to enable this goal, fos is built around the following design principles:

- OS is factored into function-specific services.

  - Applications communicate with services via message passing.

- Each function-specific service is built as a fleet of cooperating processes
  - Server processes in a fleet collaborate to provide a single OS service and communicate only via message passing.
  - Server processes in a fleet are spatially distributed across cores in a multi-core processor.
  - Server processes are bound to a core.
  - Server processes leverage distributed system techniques.

- Space multiplexing replaces time multiplexing.
  - OS runs on distinct cores from applications.
  - Working sets are spatially partitioned; OS does not interfere with application's cache.
  - Scheduling becomes a layout problem, not a time multiplexing problem.

Figure 3-4 shows the high-level architecture of fos. A small microkernel runs on every core. Operating system services and applications run on distinct cores. Applications can use shared memory, but OS services communicate only via message passing. A library layer (`libfos`) translates traditional system calls into messages to fos services. A naming service is used to find a message's destination server. The naming service is maintained by a fleet of naming servers. Finally, fos can run on top of a hypervisor and seamlessly span multiple machines, thereby providing a single system image across a cloud computer [75]. The following subsections describe the architecture of fos.

3.2.1 Microkernel

fos uses a minimal microkernel design. The microkernel only provides: (i) a protected messaging layer, (ii) a name cache to accelerate messaging delivery, (iii) an application programming interface (API) to allow the modification of address spaces and

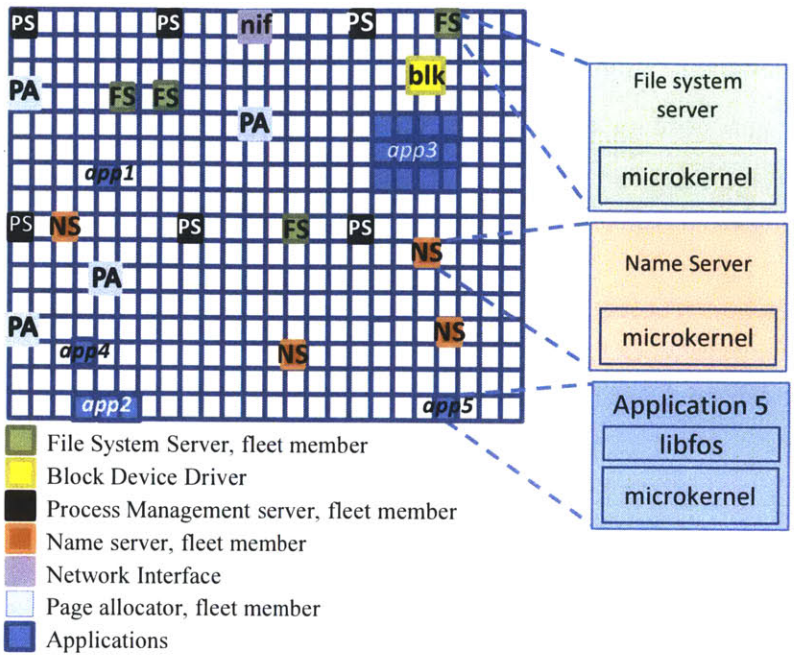

Figure 3-4: A high-level illustration of fos servers laid out across a multicore machine. This figure demonstrates that each OS service fleet consists of several servers which are assigned to different cores.

thread creation, (iv) rudimentary time multiplexing of cores to allow fos to run in a core-restricted environment, and (v) an API to allow user-level drivers to access hardware. Other OS functionality, along with applications execute in protected userspace provided by the microkernel. The fos microkernel is currently derived from the Xen Minimal OS. This was used as a starting point as fos executes as a Xen paravirtualized guest OS and the Xen Minimal OS contains examples of how to interface with the Xen hardware.

Access to the privileged functions of the fos microkernel are protected by capabilities. For example, access to the API which allows modification of address spaces and thread context is restricted to services holding a particular capability. Capabilities are also used in the messaging API to restrict sending to a mailbox.

3.2.2 Messaging

fos's basic communication mechanism is a messaging interface. The fos messaging interface is heavily influenced by the architectures it is designed for. Unlike many

previous microkernel designs such as L4 [46] or Mach [2] where messaging was designed primarily for efficient single core communication, fos messaging has been designed primarily for multicore systems which has the following implications:

- Sending core is different from the receiving core.
  - Message receive is not synchronous with message send.
- Destination core is unique.
  - Single receiver gets messages from multiple senders.
- Multicore processor is one shared trust and fault domain.
  - Messages are reliable.
  - Destinations are protected by keys (capabilities), but no heavy weight protection or security is needed.
- Messaging provides a reasonable, direct programmer interface.
  - Send is atomic once accepted.
  - Message ordering is not global, but messages are received in order from one source process to one destination mailbox. Allows some flexibility in implementation, while still allowing the programmer to reason about arrival order.

One of the largest implications of building an OS messaging system for a multicore system is that the messaging layer is optimized for the case where the sending core is not the same core as the receiving core. The URPC projects [17] has a similar constraint and one of the fos messaging implementations takes much inspiration from URPC. fos dedicates cores to applications and OS servers, therefore messages that are sent in fos go from one core to a different core. Because the receiving core and the sending core want to make forward progress and do not want to sync up to exchange a message, fos's message receive is asynchronous. fos mailboxes are a

certain size and can asynchronously receive messages from a sending core. When the receiving core is ready to receive the message, it can simply check its mailbox. The sending process also does not need to wait for the receiving process to receive the message to make forward progress. This type of messaging is in sharp contrast to L4's messaging system, which is dependent on synchronous exchange of messages, which makes sense on a uniprocessor where the send and receive can both be co-scheduled and the uniprocessor's time is best spent transitioning from the sending process to the receiving process's context.

The second insight about multicore messaging systems is that for hardware-based messaging systems or messaging built on top of a memory system which has strong locality or homing such as the TILEPro64, the destination core is unique. This is in contrast to messaging systems which were designed for uniprocessor systems, where messages were simply stored in a unified memory system. For instance in the Mach Ports messaging system, different processes can receive from a single Port. This works fine on a system where messaging has no affinity for location, but on multicore architectures, moving data to the appropriate destination is important for performance and possibly correctness in the case of hardware-based messaging networks. Therefore, the fos messaging system only allows one server to receive from each mailbox. The messaging system does enable multiple processes to send to one mailbox. Sending a message to a mailbox is a stateless operation from the viewpoint of the sender and does not require a channel setup step.

There are many different types of messaging systems which work across machines. In the general sense, network-connected computers use varying encryption based authentication schemes to ensure authentication of communication. Also, TCP is widely used on top of the unreliable IP packet protocol to ensure reliable delivery of messages. The fos messaging system has been designed primarily to work on a multicore processor. On a multicore processor, many of the challenges of message authentication and message reliability are not present. As such, fos messaging provides a reliable message transport layer and does not require the application writer to worry about lost packets, socket disconnects, and other challenges of inter-machine messaging sys-

tems. Also, because fos is being designed primarily to execute on one, large multicore processor, end-to-end encryption of message traffic is not needed because the trust model is such that message traffic will not be snooped. Last, cryptographic level authentication is not needed because the fos messaging system owns the messaging endpoints. fos does use a key to protect sending messages to a mailbox, but this is a simple key (capability) versus a heavyweight cryptographic hash.

The fos messaging system is designed to provide a reasonable, direct programmer interface in a multiprocessor, multithreaded environment. In order to support this, the sending of fos messages either are atomically sent or they signal that a send retry is needed. Atomically sending a message eases programmer complexity. Also, by returning quickly, the fos messaging API allows senders to retry sending a message or switch to different code in the event that a receive mailbox is full. Last, in order to enable different implementations of fos messaging and to allow multicore hardware to optimize around delivery order, fos messaging does not maintain a global ordering of messages. Therefore, there is flexibility around message delivery order when multiple mailboxes are being used or multiple senders are sending to one mailbox. The fos mailbox API does preserve the rule that messages sent from one sending process to one mailbox are received in the order that the messages were sent. While this puts some restrictions on implementation, it eases the burden on the programmer.

Nomenclature

This sub-section describes some of the nomenclature of the fos messaging system. The fos messaging system is built around *mailboxes*. A mailbox is an endpoint which any process can create and register with the messaging system to receive messages on. A mailbox, as the name implies, has storage which allows other processes to deposit messages into it. A process which creates a mailbox receives a pointer to a mailbox structure which can be used to receive future messages.

Mailboxes can have one or more textual names associated with them. Section 3.2.3 describes the fos naming system in more detail. The fos messaging API does not use textual names to identify mailboxes. Instead, it utilizes hashes of the textual names

called *mailbox aliases*. Other processes which want to send to a mailbox use a mailbox alias to locate the mailbox. Aliases are used instead of pointers because pointers leak information about addresses used by one process to other processes and would not work on non-shared memory systems which fos targets. Also, when compared to textual strings, aliases are shorter, fixed length, and easier to pass around.

If a mailbox is a temporary or anonymous mailbox, a name does not need to be registered for the mailbox. Instead a *canonical alias* can be used to reference the mailbox. A canonical alias is a generic alias for a mailbox which does not require explicit mailbox name space reservation.

In order to restrict processes from sending to arbitrary mailboxes, sending to a mailbox is protected by having a *capability* to protect the mailbox. A fos mailbox capability is simply a numeric key which is presented when a process is attempting to send to a mailbox. If the capability matches a capability which is on the *capability list* of the receive mailbox, the sending process is allowed to send the message. A single mailbox can have multiple capabilities on its capability list, thereby allowing different senders to have different capabilities for the same mailbox. Also, this enables revocation of a single mailbox capability without revoking all of a mailboxes capabilities.

Messaging API Overview

This sub-section gives an overview of the fos messaging API. Listing 3.1 shows the basic API for creating a mailbox. A mailbox is created with `fosMailboxCreate(...)` and destroyed by `fosMailboxDestroy(...)`. The canonical alias can be retrieved from the mailbox with `fosMailboxGetCanonicalAlias(...)`. And capabilities can be added to the capability list with `fosMailboxCapabilityAdd(...)`. Adding more aliases to a mailbox and computing aliases is discussed in more detail in Section 3.2.3.

In order to send a message, `fosMailboxSend(...)` is used. `fosMailboxSend(...)` sends to a mailbox alias which the messaging system understands how to translate to a receive mailbox. Sends to mailboxes are atomic once sent, but can return a 'retry' or

Listing 3.1: fos Messaging Mailbox Setup API

```
1  /** Creates a mailbox on the heap with data of size
2          in_buffer_size, registers mailbox with the kernel
3      @param out_mailbox_handle a newly created mailbox
4      @param in_buffer_size size of the data buffer to be created
5      @return error value in the set {FOS_MAILBOX_STATUS_OK,
6          FOS_MAILBOX_STATUS_ALLOCATION_ERROR,
7          FOS_MAILBOX_STATUS_KERNEL_ERROR} */
8  FosStatus fosMailboxCreate(FosMailbox ** out_mailbox_handle,
9                             FosSize in_buffer_size);
10
11 /** Destroys an existing mailbox, removes it from the kernel
12         and frees resources
13     @param in_mailbox mailbox to be destroyed
14     @return {FOS_MAILBOX_STATUS_OK,
15         FOS_MAILBOX_STATUS_KERNEL_ERROR} */
16 FosStatus fosMailboxDestroy(FosMailbox * inout_mailbox);
17
18 /** Retrieve the canonical alias for the passed mailbox by
19         querying the mailbox. The messaging system assigns
20         this name.
21     @param out_mailbox_alias location to deposit the
22         mailbox alias
23     @param in_mailbox mailbox to retrieve a canonical alias
24         for
25     @return error value in the set {FOS_MAILBOX_STATUS_OK} */
26 FosStatus fosMailboxGetCanonicalAlias(
27                         FosMailboxAlias * out_mailbox_alias,
28                         const FosMailbox * in_mailbox);
29
30 /** Adds a previously created mailbox capability to a mailbox
31     @param in_mailbox mailbox that the capability is for
32     @param in_capability capability to be added
33     @param in_flags flags for the capability in the set
34         {FOS_FLAG_NONE,
35         FOS_MAILBOX_FLAG_CAPABILITY_SINGLE_USE}
36     @return error value in the set {FOS_MAILBOX_STATUS_OK,
37         FOS_MAILBOX_STATUS_ALLOCATION_ERROR,
38         FOS_MAILBOX_STATUS_GENERAL_ERROR} */
39 FosStatus fosMailboxCapabilityAdd(FosMailbox * inout_mailbox,
40                         FosMailboxCapability in_capability,
41                         FosMailboxCapabilityFlags in_flags);
```

Listing 3.2: fos Messaging Send / Recieve API

```
1  /** Sends a message to a mailbox alias.
2      @param in_alias destination mailbox to send to
3      @param in_capability capability which provides
4          authority to write to the destination mailbox
5      @param in_data data to write to mailbox
6      @param in_size size of data to write in bytes
7      @return error value in the set {FOS_MAILBOX_STATUS_OK,
8          FOS_MAILBOX_STATUS_PERMISSIONS_ERROR,
9          FOS_MAILBOX_STATUS_RETRY_ERROR,
10         FOS_MAILBOX_STATUS_NO_SPACE_ERROR,
11         FOS_MAILBOX_STATUS_INVALID_ALIAS_ERROR} */
12 FosStatus fosMailboxSend(const FosMailboxAlias * in_alias,
13                          FosMailboxCapability in_capability,
14                          const void * in_data,
15                          FosSize in_size);
16
17 /** Receives a message from a local mailbox. Receives are
18         non-blocking and return instantaneously. Returned
19         buffers should be given back to the mailbox with
20         fosMailboxBufferFree quickly.
21     @param out_receive_handle the location of the received
22         message, this needs to be returned to the mailbox
23         with fosMailboxBufferFree
24     @param out_receive_size size received
25     @param in_mailbox mailbox to receive from
26     @return error value in the set {FOS_MAILBOX_STATUS_OK,
27         FOS_MAILBOX_STATUS_ALLOCATION_ERROR,
28         FOS_MAILBOX_STATUS_EMPTY_ERROR,
29         FOS_MAILBOX_STATUS_INVALID_MAILBOX_ERROR} */
30 FosStatus fosMailboxReceive(void ** out_receive_handle,
31                             FosSize * out_receive_size,
32                             FosMailbox * in_mailbox);
33
34 /** Returns the buffer to the mailbox
35     @param in_data buffer to free
36     @return error value in the set {FOS_MAILBOX_STATUS_OK,
37         FOS_MAILBOX_STATUS_INVALID_MAILBOX_ERROR,
38         FOS_MAILBOX_STATUS_BAD_FREE_ERROR} */
39 FosStatus fosMailboxBufferFree(void * inout_data);
```

'space not available' error code which may need to be looped over in order to guarantee blocking send semantics. In order to receive a message, `fosMailboxReceive(...)` is used. A receive takes a `FosMailbox` pointer as receive mailboxes can only be accessed by one process. Receiving from a fos mailbox may return a pointer to the received message or it can return an error indicating that the mailbox is empty. Finally, after a receive is completed, the received buffer must be released to the message system with `fosMailboxBufferFree(...)`. A special function is used because some of the fos messaging implementations use self-managed buffers that do not rely on dynamic memory allocation.

As can be seen from the messaging API, fos messaging does not provide the from address for received messages. The messaging API leaves it up to the user of the messaging system to put a response mailbox alias and capability in the initial request message, if a message will need a response.

Implementations

There are currently three implementations of fos messaging: microkernel messaging, shared page messaging, and an inter-machine TCP/IP tunneling implementation. The first one is a messaging layer which is implemented in the fos microkernel. The microkernel messaging layer utilizes system calls, validation in the kernel, and data copies which occur in the kernel to transfer data from one process's address space to another. This is implemented over x86-64 shared memory.

The second messaging implementation maps shared pages between two processes which communicate often. This shared page mapping is setup and torn down by the microkernel. The user-level messaging library works in concert with the microkernel to determine when it should switch over to setting up a shared page to act as a channel between two processes. By using a shared page, two processes can communicate without trapping into the kernel. These shared pages are hidden behind a strictly messaging interface. Belay details the design of the user-level messaging library in his Master's Thesis [15]. This work was inspired by URPC [17] and Barrelfish [13].

The last implementation of fos messaging is one which allows fos to be extended

across machines. Each machine runs a fos proxy server. The messaging system uses either microkernel messaging or the channel-based, shared page messaging system to communicate with the proxy server. The proxy server then encapsulates the message over TCP/IP and sends it to the proxy server on the receive computer. The receive-side proxy server then delivers the message via kernel messaging or shared page messaging to the receive process. I implemented the first fos proxy server [75] and it has since been extended and rewritten by other members of the fos group.

The fos messaging system is currently a hybrid messaging system which automatically chooses between different messaging implementations transparently to the user. In the future, we see other possible implementations of the fos messaging system on architectures which have native support for messaging such as the Tilera processor family, the Intel SCC, or the IBM Cell processor.

3.2.3 Naming

A complementary feature to the fos messaging system is the fos naming system for mailboxes. The naming system provides a hierarchical namespace for fos mailboxes. This allows symbolic names to be used for common system servers. The namespace is populated by processes which register a symbolic name for their mailbox. By using a symbolic identifier for a mailbox instead of a mailbox address, pointer, or actual processor location, mailboxes can be dynamically load balanced between servers and processes can be migrated from one processor to another while keeping mailboxes still active.

The fos mailbox namespace utilizes a textural string to represent a mailbox name. For example `/sys/fs` could be the name for the file system server mailbox. While conceptually textural names are used by the name system, internally a hash of the name is passed around when doing name lookups for efficiency. This hash is called a fos mailbox *alias*.

The fos mailbox naming system allows multiple mailboxes to be registered with the same path and alias. This enables the name server to implement a basic level of load balancing between servers. More advanced load balancing can be done by the

Listing 3.3: fos Alias Computation API

```
1  /** Computes an alias given an alias name
2          (an alias is a hash of the name)
3      @param out_mailbox_alias location to deposit the result
4          of the alias computation
5      @param in_alias_name name of the alias to compute
6      @return error value in the set {FOS_MAILBOX_STATUS_OK} */
7  FosStatus fosMailboxAliasCompute(
8                              FosMailboxAlias * out_mailbox_alias,
9                              const char * in_alias_name);
```

fleet. Advanced load balancing may be necessary if requests to a particular server fleet are stateful and a series of requests needs to all go to a single fleet member.

Originally, the fos naming system was implemented by the microkernel, but has now been implemented as a distributed fleet of name servers which execute in userspace [14]. Each process contains a name cache to reduce the need to communicate with a name server. The name cache is integrated with the messaging system to determine the actual location for a message to be sent to. There is also a name cache in the microkernel for messages that originate in the microkernel.

Modification of the mailbox namespace is protected by capabilities. In order to modify a portion of the namespace, a *mailbox alias capability* must be presented which allows modification of the namespace below, in a hierarchical directory sense, where the mailbox alias capability is for. Also, the fos mailbox naming API allows portions of the namespace to be reserved for future use in order to prevent other processes from utilizing that namespace. For example, the path `/sys` is typically reserved by the system `init` process and sub-paths under that path are given to servers that provide system level services.

Naming API Overview

This sub-section gives an overview of the fos mailbox naming API. Listing 3.3 shows the API for `fosMailboxAliasCompute(...)` which allows a process to create an alias from a textural string. This alias can then be passed into a messaging send function.

Listing 3.4: fos Name Registration API

```
1  /** Registers a direct mapping to point an alias to a mailbox,
2          in order to successfully return, the in_capability must
3          allow for modifying namespace as specified in
4          in_source_alias_name. No capabilities are needed for
5          the destination alias.
6      @param out_alias [optional] The computed alias for the input
7          name
8      @param out_capability capability returned for the newly
9          created alias
10     @param in_destination_alias alias that is being pointed at
11     @param in_source_alias_name name in the global namespace
12         which is the source of the new alias
13     @param in_parent_capability capability allowing writing to
14         parent namespace pointed at by in_source_alias_name
15     @param in_flags flags denoting what type of destination
16         alias exists in the set {FOS_MAILBOX_FLAG_NONE,
17         FOS_MAILBOX_FLAG_MULTIPLE, FOS_MAILBOX_FLAG_STATELESS}
18     @return error value in the set {FOS_MAILBOX_STATUS_OK,
19         FOS_MAILBOX_STATUS_ALLOCATION_ERROR,
20         FOS_MAILBOX_STATUS_PERMISSIONS_ERROR,
21         FOS_MAILBOX_NAME_CLASH_ERROR} */
22 FosStatus fosMailboxAliasRegisterDirect(
23                 FosMailboxAlias * out_alias,
24                 FosMailboxAliasCapability * out_capability,
25                 FosMailbox * in_destination_mailbox,
26                 const char * in_source_alias_name,
27                 FosMailboxAliasCapability in_parent_capability,
28                 FosMailboxAliasFlags in_flags);
29
30 /** Deletes an alias from the global namespace.
31     @param in_alias alias to be deleted
32     @param in_capability capability allowing deletion of alias
33     @return error value in the set {FOS_MAILBOX_STATUS_OK,
34         FOS_MAILBOX_STATUS_ALLOCATION_ERROR,
35         FOS_MAILBOX_STATUS_PERMISSIONS_ERROR,
36         FOS_MAILBOX_STATUS_INVALID_ALIAS_ERROR} */
37 FosStatus fosMailboxAliasUnregisterDirect(
38                 FosMailbox * in_dest_mailbox,
39                 const FosMailboxAlias * in_source_alias,
40                 FosMailboxAliasCapability in_capability);
```

Listing 3.5: fos Name Reservation API

```
1  /** Reserves part of the global namespace, in_source_alias_name
2          must end in '*'. in_capability must provide sufficient
3          privileges.
4      @param out_alias [optional] computed alias for input string
5      @param out_capability capability returned for the newly
6          created claim
7      @param in_namespace_name name in the global namespace which
8          is to be claimed. must end in '*'.
9      @param in_capability capability allowing writing to global
10         namespace pointed at by in_source_alias_name
11     @param in_flags flags denoting what type of destination
12         alias exists in the set {FOS_MAILBOX_FLAG_NONE,
13         FOS_MAILBOX_FLAG_MULTIPLE}
14     @return error value in the set {FOS_MAILBOX_STATUS_OK,
15         FOS_MAILBOX_STATUS_ALLOCATION_ERROR,
16         FOS_MAILBOX_STATUS_PERMISSIONS_ERROR,
17         FOS_MAILBOX_STATUS_NAME_CLASH_ERROR} */
18 FosStatus fosMailboxReserveNamespace(
19                 FosMailboxAlias * out_alias,
20                 FosMailboxAliasCapability * out_capability,
21                 const char * in_namespace_str,
22                 FosMailboxAliasCapability in_parent_capability,
23                 FosMailboxAliasFlags in_flags);
```

Listing 3.4 shows how to register and deregister a name for a fos mailbox via `fosMailboxAliasRegisterDirect(...)` and `fosMailboxAliasUnregisterDirect(...)`. Special attention must be paid to the mailbox alias capabilities used to register in the namespace. When registering a new portion of the namespace, an input mailbox alias capability must be given for a shorter path than the name being registered. Once the name is registered, an output mailbox alias capability is returned for future longer name creation or removal of the current name from the namespace. The flags passed to the name creation call denote whether the mailbox can be a one-to-many or a one-to-one name to mailbox map.

Listing 3.5 describes `fosMailboxReserveNamespace(...)` which allows reservation of the mailbox namespace without registering a mailbox. It works very similarly to `fosMailboxAliasRegisterDirect(...)`, but does not take a mailbox parameter.

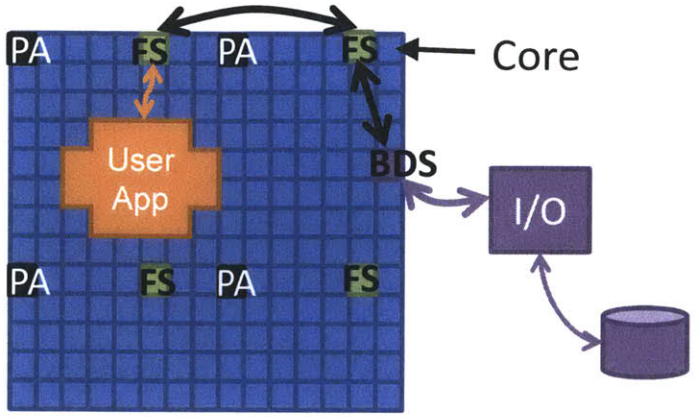

Figure 3-5: A fleet of File System Servers (FS) are distributed around a multi-core processor. A user application messages the nearest file system server which in turn messages another file system server in the File System fleet to find a file. The file is not in the second fleet member, therefore the file system server messages the Block Device driver Server (BDS) which retrieves the bits from disk.

Also, the path registered contains a '*' to denote that it is reserving all of the paths longer than the passed path.

3.2.4 Fleets

fos factors an OS by the system service being provided. One of its key aspects is that it further fine-grain parallelizes within an operating system service. It does this by taking a single operating system service or function and implementing it as a fleet of server processes.

A fleet is a spatially distributed set of server processes which communicate only via messaging and cooperate to provide a single operating system service. Each server in a fleet is bound to a core and services a local set of applications or other system servers. Fleets are designed to be elastic, meaning that the number of servers in a fleet can be increased or decreased dynamically. This elasticity is used to meet load demands on a particular service. Fleet members only communicate with each other via messaging and only communicate with applications and servers outside of the fleet utilizing messaging.

Figure 3-5 shows a fleet of File System servers (FS) which are being accessed

by a user application via messaging. As can be seen in the figure, the file system fleet is spatially distributed on four cores on the example multicore processor. In the example, the user application messages a spatially close, file system server which is its local fleet member in order to access a file. The local fleet member is not responsible for the file that is being accessed but knows how to communicate with other fleet members. It messages the File System server in the top right corner which is responsible for that portion of the file space requested. That File System server does not have the data for the file requested therefore it sends a message to the Block Device driver Server (BDS), which in turn retrieves the data from disk. The data then flows back via messaging to the file system fleet and back to the application.

fos server fleets are designed to leverage many of the ideas which have emerged from distributed and Internet scale systems. For example, fleets are constructed to make heavy use of caching and data replication to achieve good performance in a message passing only environment. One of the challenges of fleet design is how to keep shared state coherent between multiple fleet members that only communicate via messaging. Unlike true distributed systems, the communications cost on a multicore can be multiple orders of magnitude lower than inter-machine. Therefore, communication is not quite as expensive as in the distributed systems case, but reducing communication through using distributed system methods is still a great way to get higher scalability.

fos fleets are designed to provide scalable performance. One way fleets can scale in performance is by elastically changing the number of servers in the fleet in response to the load on the servers. The ability of mailboxes to be migrated through a level of name redirection enables a fos fleet to change size without disrupting the consumers of a fleet's service. Changes in size of a fleet are initiated by the fleet members instead of an external agent. When expanding a fleet, a current member of the fleet starts up a new fleet server. The new server contacts the other members either through a coordinator or through some distributed discovery protocol. After it joins the fleet, it synchronizes shared state as needed.

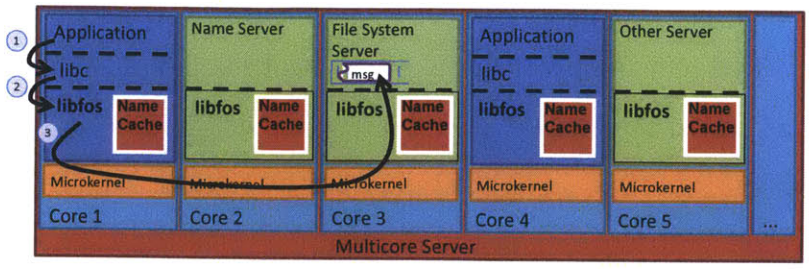

Figure 3-6: A '`fread`' function call translates into a call into `libfos`. libfos generates a message to the file system server.

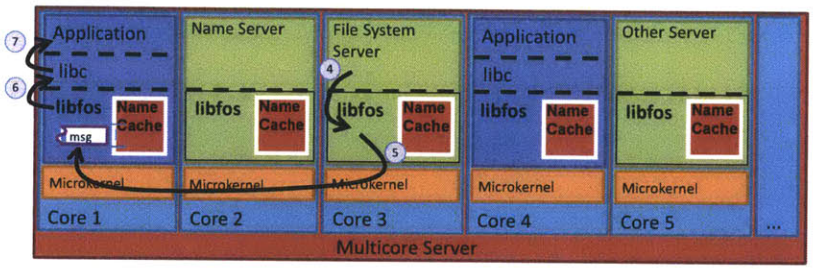

Figure 3-7: After the file system server processes the '`read`', it sends back a message with the read data. libfos translates the received message into data which is written into memory and a return code which is returned to libc and then the application.

3.2.5 libfos

fos attempts to provide a POSIX-like OS interface to enable easy porting of applications to fos. Because fos is a microkernel OS based solely on messaging, a typical application cannot make a system call into the kernel in order to receive services from the OS. Instead of executing a system call, fos user applications send messages to fos system servers. Because fos has been factored by the OS service being provided, the messages are sent to differing servers directly when they leave a user application. The library `libfos` provides the layer which translates function calls which correspond to traditional POSIX system calls into messages to function-specific servers. I implemented this translation layer along with ported FreeBSD's standard 'C' library, '`libc`', to provide fos with many of the features of legacy operating systems.

Figure 3-6 shows libfos being used to execute a system call. In this example, (1) the application starts by making a '`fread`' function call into the standard 'C' library. (2) `libc` then does a '`sys_read`' function call into `libfos`. (3) `libfos` translates

that function call into a message that it sends to the file system server. As shown in Figure 3-7, (4) the file system completes the read by hitting in its local file cache. It calls into `libfos` to send a response message to the application process. (5) The messaging library in `libfos` sends a message to the original application. (6) `libfos` on the application core demarshalls the results from the message and writes the read data into the application memory. `libfos` then returns the error code to `libc`. (7) `libc` returns the appropriate error code to the application.

3.3 Recommended Fleet Programming Model

fos does not impose one particular design paradigm on the design of fleets, but does have a set of libraries to ease in the creation of fleets. The library set consists of a Remote Procedure Call (RPC) library, a cooperative threading library, and a message dispatch library. These libraries are described in more detail in section 3.3.1. fos also does not impose requirements that application access to a fleet's members be uniform. What this means is that fleets can elect to either support a uniform API where any client can access any fleet member for service or a specialized API where a client either needs to consistently communicate with one fleet member or the client communicates with a particular fleet member as directed by the fleet. An example of where a uniform API makes sense is in a Physical Page Allocation fleet where all pages are interchangeable and transactions are stateless. In contrast, accessing a network fleet may need to be stateful because a certain fleet member may contain the state involved with one ongoing network transaction.

One recommended design for fos fleet servers is to use a request-reply model utilizing cooperative threads. While a fleet is constructed out of a set of processes, inside of a single fos fleet member or server, it is recommended that a cooperative threading programming model is used in order to simplify writing a server while providing the ability to tolerate latency. By using a cooperative threading model, the writer of a fleet is able to carefully control where and when a server yields to the cooperative scheduler. By carefully controlling where the server yields, the fleet writer

does not need to write fully reentrant code. The dispatch library provides condition variables between cooperative threads in order to protect state across yields.

A typical fos server is structured to service structured requests. When a new fleet member is created, it registers handlers for different requests that it needs to process. They typical programming model entails a run-to-completion style of programming where all of the processing for a single type of operation is written in a straight line manner. When a new request comes in from an application or other service, the dispatch library creates a new thread and executes the previously registered code. This thread handles the requested transaction, and if it needs to accomplish a long latency operation, it can elect to yield. If an active thread issues a RPC targeting a different process, it will implicitly yield. When a yield occurs, the dispatch library searches for new requests to process or response messages that have been received. The dispatch library takes care of waking the thread when the required condition variable or message response is received. These actions restart the thread. On completion of the operation, the code associated with the operation will send a message back to the original requesting process and the thread will be destroyed.

The fos threading model also supports cooperative background threads. These background threads can be used to rebalance load or do other fleet-wide housekeeping.

The recommended programming model uses several mailboxes as follows:

- New requests from processes outside the fleet
- New requests internal to the fleet
- Remote procedure call requests internal to the fleet
- Responses from internal fleet communication
- Responses from remote procedure calls

The dispatch loop imposes a static ordering on registered mailboxes and the ordering is important. The list above shows the ordering of mailboxes in increasing priority. This ordering is important because otherwise starvation and livelock situations

may exist. For instance, if new requests would be given priority over responses, then assuming that there is a steady stream of new requests, it could starve out the processing of response messages. Likewise a similar situation can turn into livelock. If we assume that one server continually polls a second server to see if a piece of state has changed. This state will only change when a response is processed, but the mailboxes have been erroneously prioritized and the constant polling will starve out the message response that is needed for forward progress.

While prototyping the dPool implementation, the above described behavior was seen. It did not result in livelock, but made a certain type of RPC traffic take thousands of times longer than it should otherwise have taken. To solve this problem and to prevent livelock, response mailboxes are given highest priority in the fos threading and dispatch model.

Another key aspect of creating a working programming model is how to limit the memory being used by active threads. Because each new request creates a new thread and those threads may sleep when performing a RPC or blocking operation, some mechanism must be in place to prevent unlimited memory from being allocated to hold the thread state. In order to prevent this, the default programming model stops processing new requests targeting a server once a fixed number of threads are active. This will in effect back pressure requests from a particular fos fleet server and prevent memory in that server from growing in an unbounded manner.

3.3.1 Supporting Infrastructure

In order to ease the creation of dPool implementations and the implementations of fos servers, a remote procedure call (RPC) library and generator along with a dispatch library were created. The RPC library handles wrapping function calls from one address space to another address space in a type-safe manner. The dispatch library is used along with a cooperative threading library to enable processing of multiple concurrent outstanding requests from a single process to a single dPool. Having multiple outstanding requests enables a dPool implementation to send and receive messages which are needed to rebalance elements, access remote elements while

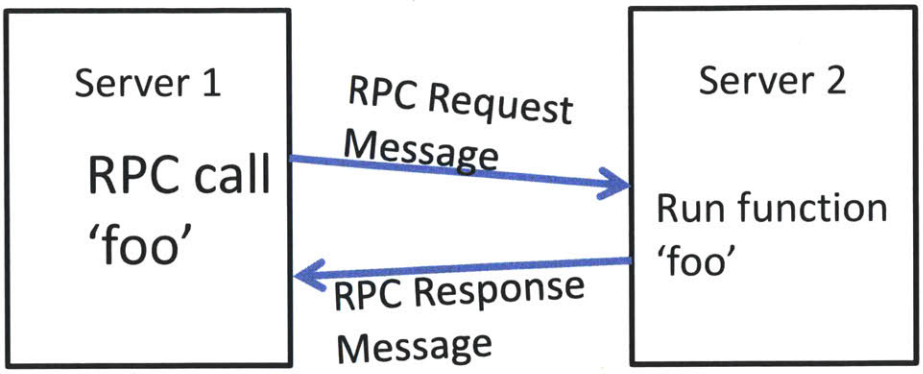

Figure 3-8: Server 1 executing a Remote Procedure Call (RPC) on Server 2.

allowing new server requests to be processed, and enable bookkeeping information to be transferred within a dPool.

RPC

Marshalling data passed over a messaging interface can be a challenge if done in an ad-hoc manner. To ease this programming challenge, I created a remote procedure call (RPC) system to make type-safe function calls between two fos servers. This utility is used within the dPool implementations along with fos servers. The fos RPC generation utility, `stubgen`, is written in python and leverages GCC-XML and the python library `pygccxml`. The `stubgen` utility utilizes standard 'C' header files which have been decorated with RPC specific attributes. GCC-XML is a program which utilizes the gcc front-end to generate meta-data about a 'C' program. Because we leverage GCC-XML, we can process any arbitrary 'C' header file with complex includes, non-RPC functions, inline functions, and complex typedefs. By generating RPC code from 'C' header files, we do not need to use a separate RPC stub generator meta-language as is used in many previous RPC generation utilities. Figure 3-8 shows the basic usage of one fos server making a RPC invocation on another fos server.

Listing 3.6 shows the prototype for a RPC callable function `bar`. All RPC callable functions return a type `_RPC` which provides error codes if there is an allocation or messaging error anywhere along the path of the remote procedure call. Input parameters are denoted with a `_IN` decoration and output parameters are denoted

Listing 3.6: Prototype of an example RPC Callable Function (bar.h)

```
1 #include <rpc/rpc.h>
2 _RPC bar(_IN int a, _IN _COPY int * b,
3     _IN _DEEP(c_ser, c_deser, c_destruct) int * c,
4     _OUT int * d, _OUT _COPY int ** e,
5     _OUT _DEEP(f_ser, f_deser, f_destruct) int ** f);
```

with an `_OUT`. By default, input parameters are passed by value, which are copied into the request message. Other more complex data-types can be passed as input parameters by the use of `_COPY` and `_DEEP` decorations. For input parameters, `_COPY` denotes that a pointer is being passed in and that the pointed to type should be copied into and sent via the message. `_DEEP` allows input parameters to be complex types such as lists or trees. `_DEEP` takes three function parameters to aid in this, namely a serialization function, a deserialization function, and a destructor function. Output parameters allow a similar level of flexibility, except that the values are returned back by reference. Because values are being returned by reference to a different server's address space, data must be copied into the result message.

One challenge with output parameters is determining when data should be freed. Basic output parameters assume that the caller of the RPC presents a pointer to a location which can be filled in with the result. `_OUT _COPY` parameters are freed by the code generated by `stubgen` on the callee's side and return a pointer to dynamic memory on the caller side. `_OUT _DEEP` parameters utilize serialization and deserialization functions similar to input parameters. The destructor function is called on `_OUT _DEEP` parameters on the callee side of a RPC.

Figure 3-9 shows the flow of files inputted to and created by the RPC `stubgen` tool. The output files can largely be spit into three categories. The first category are files which are used by the the caller side which package up parameters, send a request message, wait for a response, and demarshall return values. The second category are files shared among the caller and callee side. These are shared serialization and deserialization routines along with shared definitions for message types that are transmitted over the messaging channel. Finally, the callee side contains files that

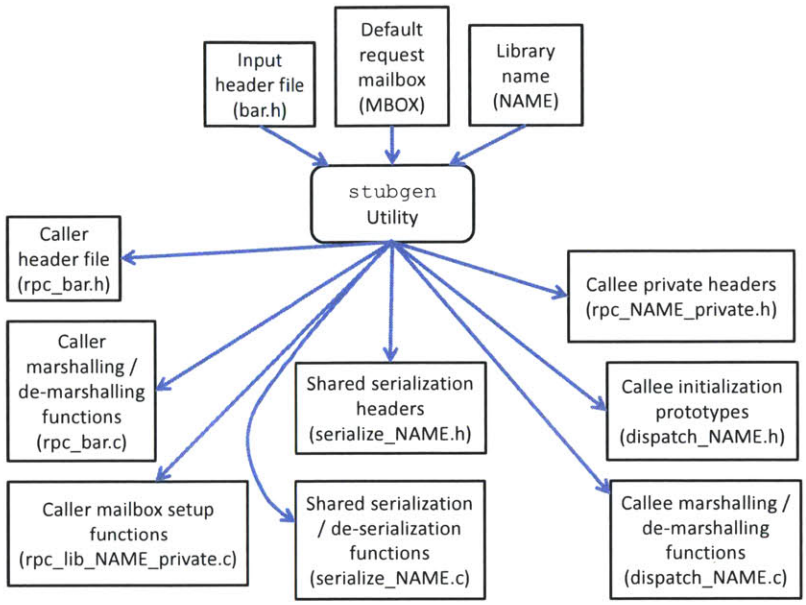

Figure 3-9: Usage flow of the RPC stub generation tool, `stubgen`

contain functions which receive the initial RPC request, format the parameters, call the function in question, marshall the return parameters, and send back the response message.

Listing 3.7 shows the code generated on the caller side of a RPC. Note that two versions of the RPC code are created, one which uses a default mailbox which prepends a `rpc_` and one which prepends a `rpcm_` which allows a mailbox to be passed into the RPC invocation. The basic flow of the RPC code on the caller side is to construct a message to send and then to initiate the send which is done in `rpc_libSend_bar(...)`. After the send, the dispatch library is called to wait for a response containing a token which was generated during the send. After the response is received, the thread which has been put to sleep is woken up and `rpc_libRecv_bar(...)` receives the message and demarshalls the output parameters.

Listing 3.8 shows the code generated on the callee side of a RPC invocation. The dispatch function, `rpc_dispatch_bar(...)`, is registered with the dispatch library to handle messages which arrive on a RPC mailbox with a statically determined

Listing 3.7: Code Generated on Caller Side

```
1  _RPC rpcm_bar(const FosRemotebox * remote, _IN int __a,
2      _IN _COPY int * __b, _IN _DEEP(.,.,.) int * __c,
3      _OUT int * __d, _OUT _COPY int * * __e,
4      _OUT _DEEP(.,.,.) int * * __f)
5  {
6      int64_t ret;
7      DispatchToken token;
8
9      ret = rpc_libSend_bar(&token, remote, __a, __b, __c);
10     if (ret < 0) return ret;
11
12     DispatchResponse * response;
13     response = rpcWaitForResponse(token);
14     if (response == NULL) return RPC_MESSAGING_ERROR;
15
16     ret = rpc_libRecv_bar(response, __d, __e, __f);
17
18     /* call caller side destructors */
19
20     return ret;
21 }
```

Listing 3.8: Code Generated on Callee Side

```
1  static void rpc_dispatch_bar(void * message, FosSize limit,
2      DispatchToken token)
3  {
4      int64_t rpc_ret;
5
6      /* args */
7      int __a;
8      int * __b;
9      int * __c;
10     int __d;
11     int* __e;
12     int* __f;
13
14     rpc_ret = rpc_dispatchRecv_bar(message, limit, &__a,
15         &__b, &__c);
16     if (rpc_ret < 0)
17     {
18         rpcDispatchReturnEarlyError(
19             &((RpcRequestMetadata*)message)->m_reply,
20             token, rpc_ret);
21         return;
22     }
23
24     rpc_ret = bar( __a, __b, __c, &__d, &__e, &__f);
25
26     rpc_ret = rpc_dispatchSend_bar(
27         &((RpcRequestMetadata*)message)->m_reply, token,
28             rpc_ret, &__d, &__e, &__f);
29     if (rpc_ret < 0)
30     {
31         rpcDispatchReturnEarlyError(
32             &((RpcRequestMetadata*)message)->m_reply,
33             token, rpc_ret);
34         return;
35     }
36
37     /* call destructors */
38     rpc_foo_free_int_STAR_(__b);
39     c_destruct(__c);
40     rpc_foo_free_int_STAR_(__e);
41     f_destruct(__f);
42 }
```

message type number. The dispatch loop then dispatches the received message to `rpc_dispatch_bar(...)`. This function demarshalls the request message into auto variables which can be passed to the call of the actual function, in this case `bar(...)`. After the function returns, the return parameters are marshalled into a message and sent by `rpc_dispatchSend_bar(...)`. Last, any dynamic data is freed.

There are several implications to fos RPC using a dispatch library. First, if the calling application is not using cooperative threads and the dispatch library, the RPC code does not cause thread switches. Instead, the RPC turns into a completely blocking operation. If the server is using the cooperative threading library, the server can process other requests. If the threading library is being used on both sides, RPC invocations can actually go from one server to another server and then the second server can make a RPC invocation on the first server and no deadlock occurs. This is useful and is actually used internally in several of the dPool library implementations. One downside to using functions which utilize the RPC library is that they may yield to other threads across a RPC invocation. The application must be aware of this and protect itself from inadvertent thread swaps across RPC function calls. This is easily done as the dispatch library has condition variables which yield in case one thread is in a critical section across a RPC invocation.

3.3.2 Dispatch Library and Threading Model

In order to ease the creation of dPool implementations and the implementations of high-throughput fos services, we created a user-level threading model and dispatch library. The threading model is based on cooperative multithreading. This was selected to ease the creation of servers as server writers can elect when the internal state of a server is safe to yield to another thread. Compared to a preemptive threading model, this removes the need for server writers to either make all code reentrant or protect global structures with atomic locks. Notional locks can still be utilized to prevent threads from interfering in critical sections, but instead of using atomic operations to accomplish these locks, simple flags accessed by reads and writes suffice.

The fos threading model was designed to enable easy writing of fos servers and li-

braries. The recommended fos server model is to write straight line, run-to-completion code which handles a complete transaction. The fos dispatch library was written to centralize much of the common code of managing the dispatch of messages based off of message types. In order to use the dispatch library, the fos server or library, such as dPool, registers a mailbox that it wants to receive messages on. Along with the mailbox, the server or library registers different message types which are contained in the beginning of messages along with functions to call when a specific type of message is received. New threads are created when new requests are received by the dispatch library. For each RPC call, a new thread is created on the callee side. When the RPC request is completed, the thread is destroyed.

The dispatch library has a token matching system for response messages. This works by allowing a thread to send a message, yield, and wait for a response. The message that is sent includes a token which is used to wake up the thread on completion. When the response message is received, the token used to find the sleeping thread, and the thread which previously yielded is woken up where it left off. An example of this can be seen in the above Listing 3.7, which calls `rpcWaitForResponse(...)` which ultimately calls `dispatchWaitForToken(...)` which waits for a token to be received.

Another feature of the threading model and dispatch library is that it enables background threads. Background or idle threads are executed when no messages are available or no currently running threads are able to run. This enables fos servers and libraries such as dPool to do background bookkeeping. An example of this can be seen in some dPool implementations which utilize background threads to rebalance elements held withing the dPool

Introducing a threading library can be good for throughput as it allows servers and libraries to hide latency and communication with other servers by processing multiple requests in parallel. Unfortunately, this also introduces some complexity. As discussed above, one challenge is that programs need to protect themselves from other threads when using the dispatch library. This is only a problem for data which is shared between cooperative threads. The threading library has convenient condition

variables to handle this problem and guard structures. The condition variables yield and put the waiting thread to sleep until the condition variable is released.

The bigger challenges come in the form of preventing too many threads being created and preventing livelock. The dispatch library starts a new thread for each new message request which is received by the server, by a RPC invocation, or by a library such as dPool. With a transaction model, if a thread messages another server and then yields, it is possible that a large backlog of threads can build up. In effect, processing new requests is always possible if there are requests pending, but completing transactions requires response messages to come from other servers outside of the control of the currently running thread. Also, because the threads are cooperative, a foreground thread can hog CPU time. To prevent the infinite growth of storage for new threads, the dispatch library limits the number of threads outstanding and stops processing new requests. This puts back-pressure on the servers or libraries trying to request services from the server that is swamped. By waiting for responses to return, the dispatch library can keep the number of active threads and memory size in check.

3.4 State of fos

3.4.1 OS Service Fleets

Table 3.1 contains a list of currently implemented fos system service fleets. These fleets are currently parallelized to differing degrees. We continue to work toward further fine-grain parallelization of these fleets. Also, some of the fleets are to be further factored. For instance, the Process Management Server fleet currently handles process setup along with virtual memory management and basic process scheduling, which ultimately will be split into separate fleets. We are working to factor this functionality into three fleets instead of the current one. The implementation of such a wide set of functionality is due to the hard work of all of the fos team members.

Fleet Name	Function	Centralized	Coarse-Grain Parallel	Fine-Grain Parallel
Block Device driver	Interfaces to Block Device	X		
Cloud Interface	Uses SOAP to launch fos VMs	X		
File System server	Read/Write Ext2 File System	X		
File System server (RO)	Read Only Ext2 File System			X
Keyboard server	Input for Xen mouse and keyboard	X		
Name Server	Maintains fos mailbox names			X
Network Stack	TCP/IP Network Stack			X
Network Device driver	Interfaces to Network Device	X		
Physical Memory Allocation	Manages physical memory allocation			X
Process Management Server	Process Startup, Basic Scheduler, and Virtual Memory Management		X	
Proxy Server	Connects messaging across multiple machines		X	
Xenbus Server	Driver for Xenbus, Xen driver initialization	X		

Table 3.1: Status of currently implemented fos fleets.

3.4.2 Applications

Currently fos implements a POSIX-like API for applications and supports multiprocessor Pthread applications. This, along with fos's system servers have enabled the porting of the following applications, all of which currently execute on fos:

- busybox (shell and basic system utilities)
- slide viewer
- lighttpd (web server)
- FFmpeg (video compression / decompression)
- wget (web client)
- SPLASH-2 benchmark suite (shared memory benchmarks)
- Portions of PARSEC benchmark suite

3.4.3 Multi-Machine fos

While fos started out as a project which applied distributed system techniques to future multicore OS design, many of the ideas are also applicable to extending fos

across multiple machines. The emergence of Infrastructure as a Service (IaaS) cloud services such as Amazon's Elastic Compute Cloud (EC2) [1] have motivated extending fos across multiple machines. This aspect of fos is not central to this thesis, but it is worth describing briefly.

Due to the fact that all fos system service fleets have been designed to only communicate via message passing and that applications only communicate with system servers via messaging, extending fos to run across a cloud was relatively easy. As described in Section 3.2.2, the fos messaging system has been extended across multiple machines via a proxy server model. Each machine in a fos cluster or cloud instance contains one or more proxy servers. The proxy server proxies messages which are destined for other machines over TCP/IP. I implemented our first proxy server, which has since been revised by other students. By making messaging transparent, fos can provide a single system view to an application. One issue that is not currently handled by the fos multi-machine effort is straddling a single, shared memory application across two machines as distributed shared memory for applications is not a current goal of this project.

We have also run fos on Amazon's EC2 and have implemented a server which interfaces with Amazon's web services API for launching new EC2 instances. Because fos system servers are elastic and programs can be migrated with relative ease, a running fos instance can grow or shrink the number of servers it is executing on dynamically. As part of our future work, I think it would be very interesting to explore how an OS scheduler's design can be modified knowing that it can add or remove computational resources in the cloud for a monetary cost.

One of the challenges with extending fos across multiple computers is bootstrapping fos and extending the naming fleet to understand multiple machines. Bootstrapping fos involves bringing up enough fos servers, such as networking, network driver, proxy, and naming, to communicate with another machine. A newly joining computer contacts the machine which spawned it and notifies the original proxy that it is now part of the running fos system. After joining, the nameserver must sync up names to provide a global namespace.

Last, we have not explored whether it is possible to use the same algorithms for distribution across machines versus inside of machines. I believe that there is a very interesting research question of whether the same techniques for building scalable intra-machine fleets work for inter-machine fleets where the communication cost is much higher.

3.4.4 Missing Functionality

While fos attempts to provide a complete operating system environment, in many ways it is still a system in development. Many of the system services are not as parallelized or distributed as we would like them to be. Also, some functionality is missing. Most notably, fos currently does not have signals. fos is currently lacking notions of user accounts and user isolation. We would like to make the fos scheduler more sophisticated. Last, we are currently implementing pipes in fos.

3.5 Challenges

3.5.1 Programming Parallel Distributed Servers

Programming parallel distributed operating system servers is one of the key challenges in the creation of fos. fos holds the operating system programmer to a high standard by requiring all communication to be explicit through messaging. Thus, the programmer must think not only about what data needs to be accessed, but where that data is located. Because messaging makes the location of data explicit, the programmer is forced to actively manage whenever communication is occurring. This requires a high degree of programmer sophistication and can make programming a serious obstacle. Libraries, distributed data structures, and common programming models can all aid in lowering the programming complexity bar.

In fos, the largest challenge to the OS programmer is managing state which needs to be shared. Resource management is an important task for an OS, and many times the easiest way to manage a resource is to use a globally shared data structure. For

example, the easiest way to manage the list of active processes is to have a table or list in memory which contains a structure for each process. Because fos is built around parallelizing low-level operating system services such as process management, it needs to tackle such problems. fos avoids using shared memory, therefore it cannot keep a single large table of active processes. Instead the fos system programmer must distribute information about processes across different members of the fos process management fleet. Keeping data coherent and distributed while still providing scalable performance to access the shared data is a serious programming challenge.

Another challenge to the fos system programmer is that the scale of and load on the system can vary widely. fos uses the approach of changing the number of servers in a fleet dynamically to react to system load. The underlying data structures used by a fos fleet must be able to handle the elastic growth or contraction of the number of processes being used. In the growth case, the data structures must distribute state to newly added servers. In the contraction case, the server must move all data to the servers which will continue running after the contraction.

3.5.2 Functionality Dependence Cycles

Constructing a microkernel operating system which factors services into different parallel fleets introduces many functional dependencies. In addition, because fos heavily utilizes messaging and naming primitives, even more dependencies are introduced through the messaging and naming systems. These constraints require the fos programmer to think carefully about the challenge of breaking dependency cycles in both services and dependencies on low-level primitives such as messaging. Factorization of the OS into many different services makes the problem of breaking functionality cycles worse in fos than in a monolithic OS. In a monolithic OS, all portions of the kernel are in the same address space and all of the interdependent parts can be co-mingled without having to break dependency cycles. Also, fos's fine-grain factorization makes the functionality cycle challenge larger than in many previous microkernel OSes where much of the functionality is in one server.

One example of this challenge can be seen in the implementation of fos's call to

allocate more memory. The call to fos's `libc` memory allocation routine, `malloc`, ultimately ends up in libfos's implementation of `sys_sbrk`. Unlike traditional operating systems which would make a system call to move the system/application break, in fos, `sbrk` must message the Process Management Server which in turn messages the Physical Memory Allocation service. One complication to this path is that fos messaging itself needs to allocate memory thereby introducing a cycle. For example, fos messaging may need to allocate memory because it may need to communicate with the name server when sending a message to look up the destination mailbox. Also, the implementation of user-space messages utilizes dynamic memory. In order to break this cycle, the destination mailbox for the Process Management Server is looked up from the name server using a fixed-size memory buffer. After this and a few other dynamic memory allocations are done to register a mailbox to receive `sbrk` responses, the fos memory allocation system switches over to using a `sbrk` based malloc. To further break the dependency on the messaging system, `sbrk` request and response messages only utilize microkernel messaging as that is guaranteed not to allocate memory thereby, breaking the dependency cycle. This is but one example of the many functionality cycles which needed to be broken in the construction of fos.

Chapter 4

dPool Design

One of the major challenges of creating fos system service fleets is sharing state between the different fleet server processes. Because all of the processes in a fos fleet only communicate via message passing, the fleet programmer in order to effectively share state needs to partition the data and devise a manner to use messages to keep the state consistent across server processes. One way to address the challenge is to factor out the shared state into a distributed data structure which manages all of the communication to keep the state consistent. A distributed data structure created in such manner can then be used by different fleets in order to leverage the work of creating such a library. In this chapter, we introduce the dPool distributed data structure, a library which provides access to shared state across multiple fleet processes for one specific shared state use case.

In order to ease sharing of state within a fos system service fleet, this chapter develops the dPool distributed data structure. The dPool data structure enables multiple fos fleet processes to share state for the case of an unordered collection of elements across address spaces. The shared state is encapsulated within the dPool data structure and the fos system programmer is simply presented a function call interface to add and remove elements from the dPool. The dPool data structure internally handles sending and receiving messages to keep the shared collection of elements synchronized.

One of the goals of the dPool is to provide scalable performance when being

accessed from different server processes within a fleet. This chapter describes the different techniques that dPool uses to increase scalability which includes the use of background threads and imprecise information.

dPool has been designed to be used by multiple fos fleets. In order to enable this, the interface provided is not tailored to storing only one particular data type. Also, dPool has been designed to integrate with different fos fleet server programming models including cooperatively threaded, sequential, and preemptively threaded models.

4.1 Semantics of dPool

dPool provides the functionality of an unordered mathematical multiset, also known as a bag. The interface to the dPool is one of atomically adding or removing elements from the multiset. There is no requirement that the elements stored in the multiset are unique and dPool does not combine elements with the same value.

dPool is a repository for storing values. As one of the design goals of dPool is to provide access to a shared resource across multiple, distinct memory address spaces, storing elements by reference inside of a dPool makes little sense. Storing *physical addresses* or pointers to global shared structures may be a valid use of dPool, but the pointers would be stored as any other data element within dPool.

dPool is agnostic with respect to the value being stored inside of an element and provides an untyped and variable size interface to elements that are contained within a dPool. This untyped and variable size interface provides generality at the expense of performance. The interface is designed in this manner to allow it to be used by the 'C' programming language. If dPool were to be used only in languages which support templating or generics, dPool could be extended to have a type-safe interface to the objects it contains. Likewise, the internal workings of dPool could be optimized around compile-time known, fixed size storage.

The interface to dPool is designed around concurrent access to the set of elements it contains. Multiple fos servers in different address spaces access the dPool through a local interface to the logically global set of elements that a dPool contains. The dPool

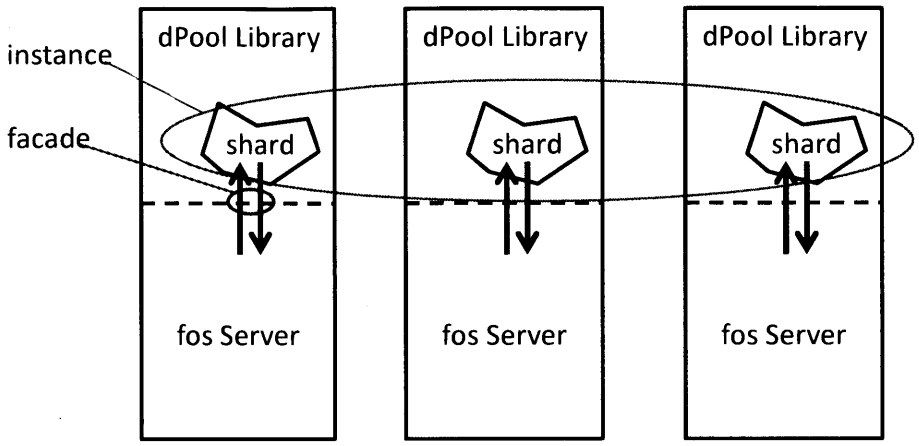

Figure 4-1: Portions of a dPool labeled. The dPool library links into the server utilizing it.

interface is designed such that the dPool interface can can have multiple instantiations of dPools active at the same time.

Introducing some common nomenclature can make different portions of the dPool system more understandable. We will name a dPool which contains a single shared set of elements, a dPool *instance*. A single dPool *instance* can be accessed from multiple fos servers. Each of these servers contains a common *interface* to all dPool *instances*, which is described below. Elements contained in a single dPool *instance* are contained in the local address space and can be distributed amongst all of the different servers which utilize a single dPool *instance*. We name the state contained in the local fos server which may contain elements from the dPool's set of elements and other bookkeeping data a *shard* of a dPool *instance*. Finally, to describe the interface local to a specific *shard* of a dPool *instance*, we name this a dPool *facade*. Figure 4-1 shows an example fos server utilizing a dPool with all of its different portions labeled.

One of the key features of dPool is that access to the elements can be concurrent across *facades* while preserving atomicity guarantees. Thereby multiple fos servers utilizing a single dPool *instance* can concurrently be adding and removing from the dPool set of elements.

While there are many data structures that provide for ordered or iterable access to a set of elements, dPool purposely avoids these requirements in order to loosen

the constraints put on dPool implementations. By loosening this constraint, no order requirement is put on the set of elements and it makes distribution of elements among the *shards* easier.

dPool has been designed to be used inside the context of multiple fos system servers. Some uses of an unordered set in an operating system include free lists and work lists. Example free lists include the free list of physical memory pages and the list of free process identifiers. Another example use of a dPool is in a work queue for a batch scheduler or a work-stealing scheduler like what appears in the Cilk [18] programming environment.

One possible implementation of a distributed data structure is to hide access to it behind a messaging interface which requires a message to be sent to a set of dedicated, centralized or distributed servers for each data structure access. The dPool interface does not rule out such an implementation, but it has been specifically designed to enable sizable storage of elements in the local *shard*. By storing elements in the local *shard*, adding and removing elements to a dPool *instance* can occur with only the cost of a function call and not the cost of a message send and receive. Also, dPool has been designed to enable sophisticated rebalancing of elements between different *shards* of a dPool *instance*. The current implementations of dPool handle management of data elements in a peer-to-peer manner, but it is possible to create *shards* which are not used through their local *facade* and only serve the purpose of increasing the aggregate storage of a dPool or to aid in balancing elements stored in other *shards* by lending another thread of execution.

dPool implementations are the product of much thought into not locking the users of a dPool library implementation into a single programming model. The dPool implementations discussed below are designed to be used by fos servers which utilize either a serial or user-level threading model. A simple extension of current dPool implementations can be made to make them thread-safe such that they could be used by preemptively multitasked servers. Not requiring preemptive multithreading by the fos servers which utilize them simplifies the design of the fos server as fos servers do not need to be reentrant. One of the challenges with not requiring preemptive mul-

Listing 4.1: Initialization for dPool

```
1  typedef struct
2  {
3      struct pool_private * private_storage;
4      uint64_t object_number;
5      FosRemotebox * mailbox;
6  } Pool;
7
8  /* Creates a new Pool
9  */
10 Pool * poolCreate();
11
12 /* Creates a new Pool Facade
13    master_mailbox_name is the mailbox of a preexisting dPool
14        fleet which we are to join.
15    returns a pointer to a Pool struct */
16 Pool * poolInitFacade(FosRemotebox master_mailbox_name,
17    uint64_t object_number);
```

tithreading is that a dPool may need to send and receive messages between different *shard*s without the aid of the server which is using the dPool library. This and other challenges are discussed below.

The dPool interface has been implemented by a set of different implementations which each have different properties as are described in Section 4.4. The common interface allows a fos server to choose and later change the dPool implementation utilized in order to meet a particular server's needs.

4.2 Interface

This section describes the 'C' language interface to dPool. This interface is the *facade* through which fos servers can initialize, access elements, adjust locality, and elastically resize a dPool.

4.2.1 Initialization

Listing 4.1 presents a source code listing to initialize a dPool. The first server to initialize a dPool calls `poolCreate()` to create a new dPool. The initialization of the dPool on the first server is different than subsequent servers because it must initialize a mailbox and does not have any other dPool *shards* to contact. Subsequent servers initialize the dPool by calling poolInitFacade with a mailbox which was created by the initialization of the first dPool *facade* and the instance number which was created. The `master_mailbox_name` and `object_number` must be passed out of band relative to the dPool. It is assumed that fos servers utilizing a dPool will already have some way of communicating with other servers in the fleet in order to sync up such information. The `object_number` parameter is a mechanism to allow multiple dPool *instances* to co-exist in the confines of a single 'C' programming language namespace.

On completion of `poolCreate()` or `poolInitFacade(...)` calls, the pool is ready to use. These functions return a pointer to a newly created dPool (`Pool *`) object.

4.2.2 Element Access

Listing 4.2 shows the different ways to access elements in a dPool. The interface is quite simple as to add an element, `poolAdd(...)` is used and to retrieve an element, `poolGet(...)` is used. Note that `poolAdd(...)` copies `size` bytes of the value pointed to by `value` into its internal storage. `poolGet(...)` copies a found value into a buffer pointed to by `found_value`. If no value is found or the found value is larger than the `size` passed to `poolGet(...)`, then an error is returned. `poolGet(...)` is guaranteed to return an element of the dPool if any element exists in the dPool. Also, `poolGet(...)` is guaranteed to return a unique element to each calling *facade* if concurrent access is occurring.

Last, `poolSize(...)` returns an estimate of the number of elements in a dPool. An estimate is utilized because it is challenging to provide exact information. Each dPool *shard* contains the number of elements that each *shard* contains and this information can be accessed in constant time for the local *shard*. A global estimate must

Listing 4.2: Element Access for dPool

```
1  typedef int PoolSize_t;
2
3  /* Inserts an element into an initialized Pool.
4      Returns 0 on success, -1 on error */
5  int poolAdd(Pool * po, int size, void * value);
6
7  /* returns a random pool element.
8      Returns -1 if not found,
9      returns -2 if object to be returned is larger than size
10     returns the size of the value returned.
11     If found object is larger than size, the
12     output is not filled in. */
13 int poolGet(Pool * po, int size, void * found_value);
14
15 /* returns the number of elements in the pool
16     returns -1 on error, 0 on empty, otherwise returns number
17     of elements in pool.
18     To preserve performance, poolSize is not atomic. */
19 PoolSize_t poolSize(Pool * po);
```

contact the other *shards* in the system in order to get an accurate count of elements in the entire dPool *instance*. While this information is being gathered, concurrent access to the other *shards* is proceeding which would cause the exact result to be out of date. One way to have precise information, is to stop adds and gets from occurring while the size of the pool is calculated. Another approach would be to use time-stamp based model where the `poolSize(...)` request would return the size of the dPool atomically from some time in the past. In order to prevent these complexities and performance impact, it was decided that this interface would return an estimate to preserve performance of dPool implementations. Also, for usage models envisioned, this was the least important interface to the dPool.

4.2.3 Locality

Some dPool implementations can use locality to optimize how dPool *shards* communicate and balance resources. The basic model is that each server using a dPool would

Listing 4.3: Locality Interface for dPool

```
1  typedef struct
2  {
3      char data [64];
4  } PoolLocation;
5
6  /* returns an opaque location structure */
7  PoolLocation * poolFindLocation(Pool * pool);
8
9  /* set a distance between pool_location_from and
10     pool_location_to with metric.
11     Note that the default distance is 10,000 */
12 void poolSetLocality(Pool * pool,
13     PoolLocation * pool_location_from,
14     PoolLocation * pool_location_to, int metric);
```

query its local *facade* to determine its 'location'. Then via out of band means, the server determines the distance between each pair of servers. Last, the server sets the distance between any two given dPool *facades*. The setting of distances can be done on any dPool *facade* as they are all internally connected via messaging. The locality interface is just an optimization hint and does not override guarantees provided by the element access interface.

Listing 4.3 provides a full listing of dPool's locality interface. The `PoolLocation` is specific to a particular implementation of a dPool and should be regarded as an opaque structure. Last, if a server is migrated, the locality metric can be updated via a call to `poolSetLocality(...)`.

4.2.4 Elasticity

Elasticity is the ability for a dPool to increase and decrease dynamically the number of servers using a single dPool *instance*. In order to expand a running dPool, `poolInitFacade(...)` is used as described above. In order to reduce the number of *shards* in a dPool *instance*, `poolShutdown(...)` is used as described in Listing 4.4. If there is more than one *facade* active in a dPool *instance*, the shutdown will shutdown the local *facade* and push any important state and elements to other *facades*. If this is

Listing 4.4: Shutdown Interface for dPool

```
1 /* Shuts down the pool object and flushes
2     any dirty state to other pool shards
3     returns -2 if this server is unable to shutdown.
4     returns -1 if this is the last pool server in a instance
5         to be shut down.
6     returns 0 if local facade shut down successfully. */
7 int poolShutdown(Pool * po);
```

the last *facade* being shutdown, a special return code is returned such that the server can know that this is the last dPool *instance* and that the remaining elements have been destroyed.

4.3 Elasticity

One of the important aspects of fos service fleets is that they can grow and shrink in size. This is done such that they can respond to load. They can also be shrunk down to use fewer cores which may ultimately save energy or allow reuse of the cores. Because dPool is designed to be used by fos servers that can elastically change the number of servers providing a service, dPool must also elastically add and remove *shards* from an executing dPool *instance*. Section 4.2.1 and Section 4.3 describe the external elasticity interface from the server's perspective. This section describes how dPool *shards* join and leave a dPool *instance*.

dPool implementations do not currently try to optimize for growing or shrinking the number of dPool *shards* in a *instance*. Current dPool implementations use a master to track the growing or shrinking of a dPool. When a dPool is initialized, the first dPool *shard* is used as the master. After that, when a new *facade* is initialized, it contacts the master dPool *shard* to signal that it wants to join the dPool *instance*. The master keeps a list mailboxes for all registered dPool *shards* which it pushes to the requesting *shard* along with a new *shard* number. The master guarantees that the list sent to the joining *shard* is complete as it does not allow multiple registration or removals to happen concurrently. After the new *shard* has received the list of all

other *shards*, it messages the other dPool *shards* so that it is added to their list of active *shards*. After it has registered itself with all of the other servers, it internally marks itself as healthy and begins processing requests.

When a dPool *shard* is shutdown and leaves a dPool *instance*, it begins by not accepting any new add requests from other dPool *shards*. Next it pushes all of its current elements to other dPool *shards*. Currently, it does this by pushing its elements in a batch round-robin fashion to the other *shards*.

Now the dPool *shard* removes itself from all of the other *shards*' peer lists. It does this in two phases, first, it removes itself from the peer lists on all of its peer *shards*. When the leaving dPool *shard* is removed from the peer list, it is put on a list of pending removals. A *shard* will not start any new transactions to the removed *shard*, but there may be outstanding transactions to the shard which must be given a chance to complete. Therefore, this *shard* is in effect reference counted while it has pending transactions in flight to it. The second phase involves sending a second request to every peer *shard*. This second request waits for the reference count to reach zero, frees the memory for the peer list element, and then returns. At this point, the dPool *shard* that is leaving the dPool *instance* is no longer referenced by any other *shard* and contains no elements. It is now free to deallocate memory and return.

Currently, fos dPool implementations assume that the master server is the last server to be removed from a dPool *instance*. This is because there needs to be some way of contacting a master mailbox to join a running dPool *instance*. With modifications to the API, it could be possible to allow the dPool master to be transferred to another dPool *shard*.

dPool was designed for performance of adding and retrieving elements. One problem with current dPool implementations is that each *shard* contains a list of each other *shard*. This optimizes add and get performance, but slows down adding and removal of new dPool *shards* to be a $O(n)$ time operation. Also, the per *shard* peer list of other *shards* uses $O(n^2)$ storage.

4.4 dPool Implementations

In this section, we describe the different dPool implementations and algorithms used inside of them. This section will be referred to in the results chapter when these different dPool implementations are used in the context of a physical page server.

4.4.1 Centralized Storage

The most basic implementation of dPool contains a distributed interface, but does not distribute the storage of elements as the name dPool implies at all. The centralized storage implementation stores all of the elements contained in a dPool *instance* in the first dPool *shard* created. Subsequent *shard*s simply message the first *shard* when an element is requested from them through their respective *facade*s. When an element is added to a *shard* which is not the first *shard*, the first *shard* is messaged and the element is placed in the first *shard*'s storage. In effect, all dPool *shard*s which are not the first *shard* act as a *facade* for the first *shard* and use RPCs over fos messaging to communicate with the first *dPool*.

4.4.2 Distributed Storage

One step up in complexity from a centralized storage dPool implementation is one which allows each *shard* to contain a local list of elements, such as in the distributed storage dPool. When an element is added to the dPool, it is added to the local *shard*'s list. When an element is retrieved from the dPool, if an element is available in the local *shard*, it is removed from the head of the local *shard*s list and returned. If there are no elements available in the local *shard*'s list, the *shard* will contact the other *shard*s looking for an element. If no element is found, the local *shard* returns the appropriate error code. In order to guarantee fairness when a local *shard* runs out of elements, the *shard* that it first contacts when requesting an element rotates. Each *shard* keeps track of the last dPool *shard* that it contacted last and each time it goes to start contacting a new *shard* it rotates to the next *shard*.

4.4.3 Distributed Storage Bulk Transfer

One of the problems with the distributed storage dPool implementation described above is that unless a dPool's *shard* has elements added locally, a local *shard* will never have any elements in its list. Also, if the local list is empty, every request from a local *shard*, will kick off a worse case $O(s-1)$ number of messages in order to find an element, where s is the number of *shards*. In order to reduce communication, the distributed storage bulk transfer dPool implementation functions very similar to the distributed storage dPool implementation, but pulls multiple elements in one message. Nominally, when an element is requested from a *shard* and the *shard*'s local list is dry, the *shard* will attempt to pull 50 elements from another *shard*. If one or more elements are available when requesting elements from a different *shard*, up to 50 elements are transferred and the requesting *shard* stops and returns one of the found elements. If the other *shard* contains no elements, the *shard* which originally requested an element will go onto the next *shard* therefore, the worst case communication cost to find an element is still $O(s-1)$. While the worst case communication cost has not changed, the probability that a local *shard* will need to make a remote request will have gone down by a factor of 50. Also, this algorithm makes it more likely that other *shards* will contain elements thereby reducing the probability that a *shard* will contact another *shard* and that second *shard* is empty.

4.4.4 Distributed Storage Bulk Transfer with Background Pull

We expand on the distributed bulk storage algorithm by adding background threads. dPool is built within a cooperative threading framework so background threads only operate when there are idle cycles. This is nice because it eases the programming model as the dPool implementation does not need to guard against preemption. Also, only truly idle cycles are harvested for background optimization. The distributed storage bulk transfer with background pull implementation extends the distributed bulk storage algorithm implementation with the addition of a background thread

which pulls elements from other *shards*. In order to rate limit background pulling, the background pull thread only executes once out of every 100 idle thread scheduling events. Once it is determined that a pull should occur, the local *shard* only attempts to pull elements from other *shards* when the local list contains fewer than 1024 elements. The implementation attempts to pull 50 elements at a time from another *shard*. The *shard* which is chosen to be pulled from is rotated amongst all of the other *shards* in the system in a round-robin manner. If the *shard* being pulled from has no elements, the background thread returns and waits to be rescheduled. This implementation similarly directly pulls 50 elements from another *shard* in response to a local request.

4.4.5 Distributed Storage Bulk Transfer with Background Push

The distributed storage bulk transfer with background push implementation extends the distributed storage bulk transfer implementation with the addition of a background thread. In contrast to the background pulling thread discussed above, the background push thread pushes elements from one dPool *shard* to other dPool *shards* instead of pulling elements. The background thread is only activated every 100 idle thread scheduling events and like the background pull thread is only scheduled when the processor is otherwise idle. The background push thread pushes elements only if the local dPool *shard* contains more that 20,000 elements. The background thread pushes 50 elements at a time and pushes them in a round-robin manner to other *shards*. This pushing is done indiscriminately of how many elements the other *shards* contain. One downside to this approach is that a *shard* may end up pushing elements to a different *shard* which already contains more elements. Also, it is possible for *shards* to endlessly push elements between each other thereby increasing the amount of useless work. One bright side of using a background push thread is that the background thread only pushes when there is no critical work to be done. Therefore if a particular dPool *shard* is busy, it will not be creating extra traffic and pushing away potentially useful elements.

4.4.6 Distributed Storage Bulk Transfer with Background Push and Element Estimation

The distributed storage bulk transfer with background push and element estimation implementation extends the above background push implementation by adding intelligent decision making about which *shard*s should push to which other *shard*s. This intelligent decision making works by having each *shard* keep an estimate of the number of elements that each other *shard* contains. This is done through a lazy update system. The lazy update system is triggered from the background idle thread in each *shard*. This update system works by each *shard* checking whether the number of elements currently contained in the *shard*, a constant time operation, is either 500 elements larger or smaller than the last estimate which was sent. If the current, local pool size is 500 elements different from the last sent estimate, the lazy update thread notifies the master *shard* of the number of elements that the *shard* now contains. The master *shard* collects new estimates and periodically broadcasts the estimate updates to all the other *shard*s. There is an interesting trade-off between how often estimates should be transferred, what the threshold should be before transfers occur, and the quality of the information. More up-to-date information requires more messaging overhead.

Now that each *shard* has an estimate of how many elements all of the other *shard*s contain, the background push thread will only push to other *shard*s which have estimated fewer elements. The implementation only pushes to other *shard*s which have 2,048 or fewer estimated elements than what the local *shard*'s list contains. This modification to the algorithm severely limits the number of elements that bounce back and forth between *shard*s.

Chapter 5

dPool Service Integration

dPool is designed to be reusable by multiple fos OS services. This chapter describes two different uses of dPool. The first is inside the Physical Memory Allocation server fleet where it is used as the primary data structure for keeping track of free physical pages. It is also used by the Process Management Service fleet where it is used to dole out unique process identifiers.

5.1 Physical Memory Allocation

In any OS, allocation of physical memory is an important basic system service. The fos Physical Memory Allocation fleet is used to allocate physical memory pages within a physically shared address space. fos has been designed to run on future multicore processors which have a shared global physical address space, multiple shared physical address spaces, or independent address spaces per core. fos explores how to manage such architectures while only internally using messaging. The Physical Memory Allocation fleet is designed to manage physical memory for architectures which have a shared global physical address space or multiple shared physical address spaces. In order to manage memory on a system with multiple physical address spaces, multiple, independent Physical Memory Allocation fleets can be used.

The fos Physical Memory Allocation (PMA) fleet provides a uniform interface for

Listing 5.1: Library Side Interface to the Physical Memory Allocation Fleet

```
1  /* returns the physical address that is the start of a page
2      that can be used. Returns 0 if no pages available. */
3  physaddr physPageAlloc();
4
5  /* release the page to the physical memory allocation
6      system */
7  void physPageFree(physaddr addr);
8
9  /* frees an array of n pages passed in an array. */
10 void physPageFreeBatch(int naddrs, physaddr * array);
```

each member of the fleet. The interface is only accessible via messaging and assumes that the server asking for pages is a trusted entity. Like most fos system services, the Physical Memory Allocation fleet provides a client library to ease the usage of the Physical Memory Allocation fleet by other fleets. Listing 5.1 shows the library-side interface to the Physical Memory Allocation fleet. These library functions are thin veneers over messages which are sent from the user of the library to the PMA fleet. One interesting aspect of the library is that it uses the threading model and dispatch library, if available, or else it simply blocks waiting for message responses. Also, the library takes advantage of sending multiple, parallel messages to the PMA when the `physPageFreeBatch(...)` function is called with a sufficiently large enough number of pages to be freed.

One interface explicitly missing from the interface to the Physical Memory Allocation fleet is an interface to ask for differing size pages or chunks of memory. As fos is designed for machines with flat memory spaces (64-bit), there is no need for a buddy allocator of physical memory as all pages are created equal and are interchangeable.

Currently the Process Management Server is the only server fleet that communicates with the Physical Memory Allocation fleet. The Process Management Server fleet handles the creation of new processes for which it needs physical memory pages. It also currently serves the role of managing a process's virtual memory and handles requests for dynamic memory through a message based `sbrk` request.

In order to implement the Physical Memory Allocation fleet, the fleet uses a

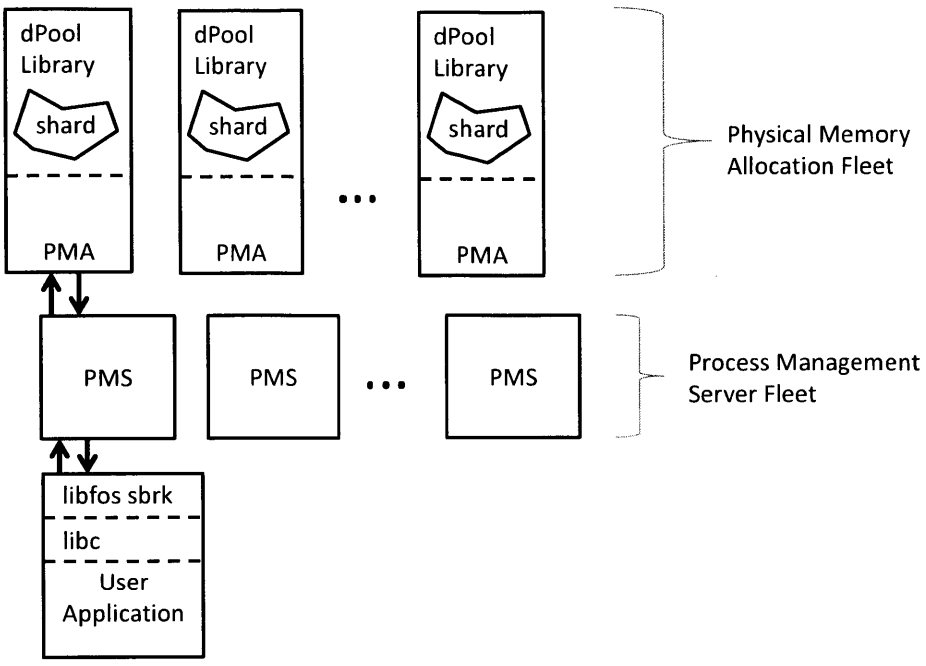

Figure 5-1: A User application allocating memory. Messages shown as arrows.

dPool internally to store all of the free pages. The PMA currently uses a single dPool instance where all the elements are four bytes in size. The PMA stores physical frame numbers, therefore with 4KB pages, the PMA currently is limited to managing 17TB of physical memory. To extend this requirement, a simple change can be made to the page frame number type definition.

When the first PMA server starts up, it executes a protected system call into the microkernel to get physical pages from the microkernel. After the startup series of calls, allocation of physical pages is handled by the PMA.

Figure 5-1 shows the basic usage of the Physical Memory Allocation fleet to service a user application's request for memory. The application starts out by calling `malloc` in `libc`. `libc` calls `sys_sbrk` inside of `libfos` which in turn sends a message to the Process Management Server fleet. In the future, memory allocation may be broken out of the Process Management Server fleet into a Virtual Memory Server fleet. The Process Management Server, then calls into the library-side interface for the PMA which in turn sends a message to the PMA. The PMA then gets a page out of the dPool *shard*. The *shard* may need to communicate with another *shard* stored in a

different PMA server. The dPool returns a frame number or an error code to the Process Management Server. The Process Management Server then maps the page into the user process and returns the new memory break (brk) location. The Process Management Server also requests pages from the PMA when creating or destroying processes.

The current implementation of the PMA does not take manage multiple Non-Uniform Memory Access (NUMA) memory nodes. There are several ways to extend the PMA to allocate memory from different NUMA memory nodes. One way is to have the PMA fleet utilize a dPool instance per NUMA node. Another way that we have considered is to allocate pages for the local NUMA node in the dPool shard closest to the memory controller with that NUMA node. While dPool does rebalance elements (pages), locality of pages will be preserved to a large extent inside of dPool. Pages are most likely to be changed to different regions of the chip when the dPool becomes low on pages which is the same behavior of many other NUMA physical page allocators.

The PMA does not currently take into account page coloring as the architectures that we are currently running fos on do not exhibit advantages to page coloring in the L1 or L3 caches. For instance on a Intel Nehalem processor, the L1 cache is 8-way set associativity and are small enough that the maximum size of a cache way is the size of the smallest page size. The L3 cache is 16-way set associative and shared between multiple cores therefore making it very challenging to page color. This leaves the L2 where page coloring could help, but it too is 8-way set associative decreasing the benefit. Last, by adding page coloring, there is higher overhead in the allocation of a page.

5.2 Process Identifier Allocation

The Process Management Server manages the creation and destruction of processes. One important aspect of process creation and destruction is the allocation of process identifiers (PIDs). A PID allows different system utilities to reference a process by

a simple numeric. In order for PIDs to be useful, they need to be unique. The fos Process Management Server (PMS) fleet is a distributed fleet. Because PIDs are allocated from one global identifier space but the Process Management Server fleet is distributed, a way to keep PID allocation and deallocation coherent across multiple Process Management servers is needed. The PMS utilizes a dPool to manage PID allocation and deallocation.

The use of a dPool being used for PID allocation in this section makes some assumptions about PIDs. This section assumes that it is advantageous to allocate PIDs from a fixed set of PIDs. One advantage to this is that this can keep PIDs in a smaller range than if PID numbers are only used once. Also, by using PIDs from a fixed set, the size of the PID can be smaller therefore utilizing less storage. For interactive systems it can be easier to type in PIDs from a smaller fixed size pool than if they are long random numbers.

The Process Management Server fleet was written by a different author than the author of dPool and this thesis. dPool was easily integrated into the Process Management Server fleet and shows a second use of the dPool inside of the context of fos system service fleets.

The initial Process Management Server creates a dPool *instance* and then adds all of the available PIDs to it at startup. Then, as processes are created or destroyed, PIDs are added or removed from the dPool. All of the servers in the Process Management Server fleet utilize one dPool *instance*, therefore guaranteeing that no two concurrently executing processes are given the same PID. PIDs can be added or removed from the dPool concurrently as each Process Management Server fleet member can operate on its own dPool *shard*. The Process Management Server takes advantage of the scalable algorithms and rebalancing of elements that the dPool provides, thereby reducing the probability that any single *shard* will run dry of PIDs.

By preloading a dPool with all of the available PIDs, we can see that dPool can be used for more than a simple free list. Instead, it can be used for atomic allocation of entries out of a common list. In the current implementation, we load the dPool with one million PIDs which is the same number of PIDs as standard Linux systems.

The usage of dPool for PID allocation meets the current needs of the fos PMS fleet, but there are several extensions which we have been investigating. One is that it may be desirable for system security reasons to prevent quick reuse of PIDs. One way to prevent this is to use two dPool *instances* within the PMS. At the beginning of time, the first dPool is loaded with all of the PIDs in the system. Allocation then proceeds by allocating PIDs from the first dPool and when a process terminates, the PMS inserts the PID into the second dPool. When all of the PIDs from the first dPool are utilized, all of the PMS fleet members agree to switch the usage of the two dPools. It now starts allocating out of the second dPool *instance* and inserting terminated PIDs into the first and the cycle continues. This use of two dPool *instances* forces the PMS to allocate all PIDs before reusing a PID.

Last, the PMS use of dPool utilizes $O(n)$ memory, where n is the number of PIDs available. In comparison, the Linux PID allocation mechanism utilizes a bitmap which uses $O(m)$ storage, where m is the total maximum number of PIDs in a system. The constant factors used in Linux are smaller due to using a bitmap versus a list element, but the asymptotic storage is the same.

I foresee many similar uses of dPool within fos system servers. One example is the allocation of outbound socket numbers which need to be allocated across many distributed network stack fleet members. dPool may also be a good fit as a work pile scheduler. Jobs can be added to the work pile concurrently and they can be removed concurrently with a dPool.

Chapter 6

dPool Performance Analysis

This chapter explores the scalability of the dPool data structure. We measure dPool as integrated into the Physical Memory Allocation server fleet. Different micro-benchmarks are utilized to determine the best algorithm and the best partitioning of data to use under differing loads.

6.1 Experimental Setup

The results presented in this chapter are all gathered on the current version of fos executing on a 48 core (quad socket 12-core) AMD server. The server has four 1.9GHz AMD Opteron 6168 processors totaling 48 cores and 64GB of RAM. fos runs as a paravirtualized OS under Xen [11]. The experiments in this section were collected with fos executing as a DomU under Xen 4.0.1 running with a Linux Dom0 running Linux version 2.6.31.13.

In order to stress the dPool used by the Physical Memory Allocation service, a test harness was created to simulate the traffic of many applications simultaneously allocating and mapping pages. As part of the test, the test harness client program requests a 4KB page from a PMA server, maps the page into the test harness's address space and zeros the contents of the page. At the end of the test, the test harness frees the pages back to the PMA server. This, in effect, models the actions of the Process Management Server and a user application requesting pages from it. The results

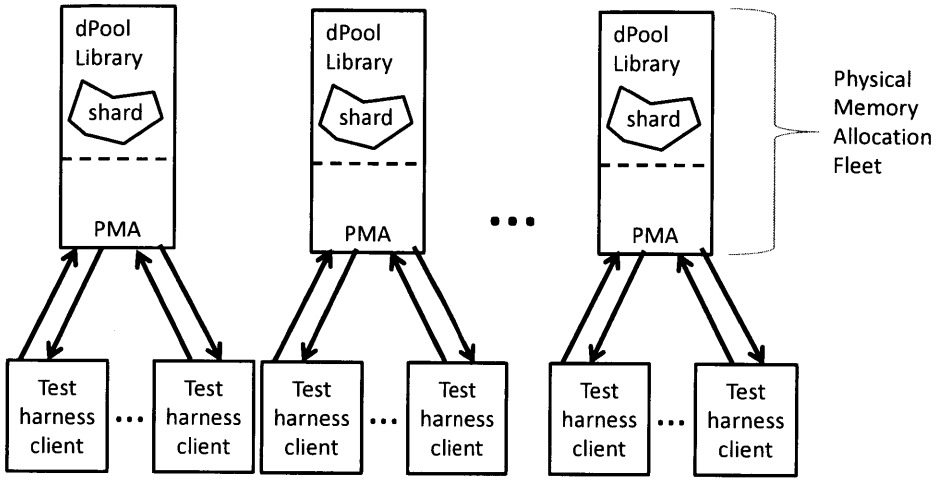

Figure 6-1: Primary Test Setup. Multiple test harnesses can connect to a Physical Memory Allocation server. Multiple Physical Memory Allocation servers use one dPool *instance* to manage the list of free pages.

presented below use the test harness along with the Physical Memory Allocation fleet running inside of a functioning and booted fos system. Figure 6-1 shows a fleet of PMAs serving multiple test harness clients.

A test harness is used to test the dPool that is integrated inside the Physical Memory Allocation server to best isolate the scalability of the dPool inside of a functioning fos system service fleet. By using a test harness, parameters such as placement of servers, placement of clients, rate of request, and assignment of servers to clients could all be closely controlled. The test harness is also able to drive traffic that is more demanding than could otherwise come from a traditional application.

By using a test harness to test dPool scalability, fewer processors were used for the load generation than if an application and PMS were used to generate load. This has allowed us to test larger configurations of the dPool before running out of processors on the test system. Last, by using a test harness, the development of the Process Management Server fleet and the Physical Memory Allocation fleet utilizing a dPool could be decoupled. This has been especially useful as the PMS is currently less fine-grain parallelized than the PMA fleet, and it has been developed by another student.

6.2 Workload Description

We use the test harness to drive different distributions of page allocation and mapping against varying numbers of PMA servers. Each test harness client communicates with only one PMA server. The test harness clients are distributed amongst the servers. If the number of clients is less than the number of servers, then not all servers will have a client communicating with it. If the number of clients is not an integral multiple of number of servers, they are distributed between the different servers, but some of the servers will service strictly one more client than other servers.

Unless otherwise noted, all of the free pages in the page pool are initially centralized on the first PMA server. This is because the PMA has been designed to elastically grow and shrink the fleet size. Therefore, at the beginning of each test, all of the free pages in the system are added to the first PMA server and either the other servers in the fleet need to request pages from the first PMA server or the background threads rebalance the pages.

In each test, the overall page pool consists of the number of servers times 65,536 pages, which amounts to 256MB of memory per server. For each of the tests below, the total number of pages allocated is slightly less than the total number of pages in the system for a given test. This is done such that the PMA server fleet does not run dry, but we do stress the low page case. The low page case is the most challenging algorithmically, as it is possible that the dPool contained in a PMA server may need to contact every other PMA server in the system to find a free page.

We test the dPool used inside of the PMA service with four, primary traffic distributions being driven by test harness clients requesting pages. The first distribution is that of a uniform distribution. In the uniform distribution, each client requests an equal number of pages. As discussed above, each client requests slightly less than their equal share of the pages. For the uniform case, each client requests the number of servers times 1024 fewer pages than their equal share of pages.

The second distribution is a non-uniform distribution. It is an increasing linear distribution. We test non-uniform distributions to see how dPool scales when

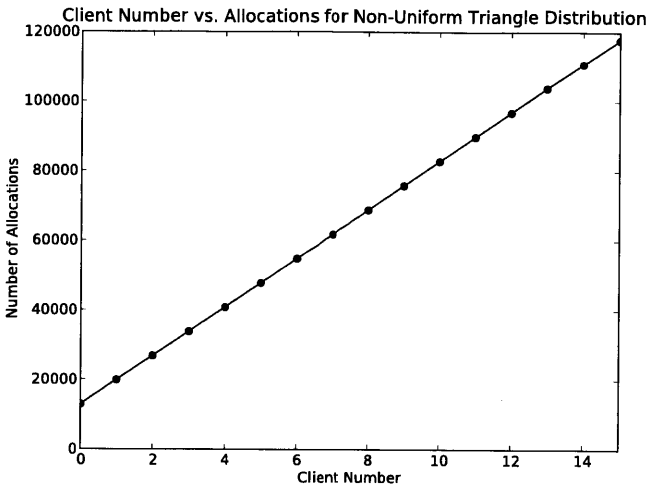

Figure 6-2: Number of allocations completed by each client in the non-uniform, triangle distribution with 16 clients and 16 servers.

presented with uneven load. As the client number increases, the number of pages allocated and mapped increases. The distribution starts out with 20% of the even share of pages allocated and linearly increases such that the last client uses 180% of the even share. So as to not allocate all of the pages, each client allocates 256 fewer pages than the nominal number. This works out to $pages = (totalPages/numClients) * (0.2 + 1.6 - (1.6 * ((numClients - clientNum - 1)/(numClients - 1.0)))) - 256$ as shown in Figure 6-2 for 16 clients and 16 servers. We will refer to this distribution as the Non-Uniform Triangle distribution.

The third distribution is a non-uniform, bimodal distribution. It alternates with every other client either allocating 180% of the nominal pages: $pages = (totalPages/numClients) * (1.8) - 256$ or 20% of the nominal pages $pages = (totalPages/numClients) * (0.2) - 256$. This is shown graphically in Figure 6-3 for 16 clients and 16 servers. We will call this the Non-Uniform Bimodal distribution.

The fourth distribution is a time varying non-uniform distribution. This distribution begins by running the same distribution as the Non-Uniform Bimodal. After all of the clients have finished executing the initial bimodal distribution, a barrier is executed and the clients begin executing a distribution where the clients which were previously requesting a large number of pages now request few pages

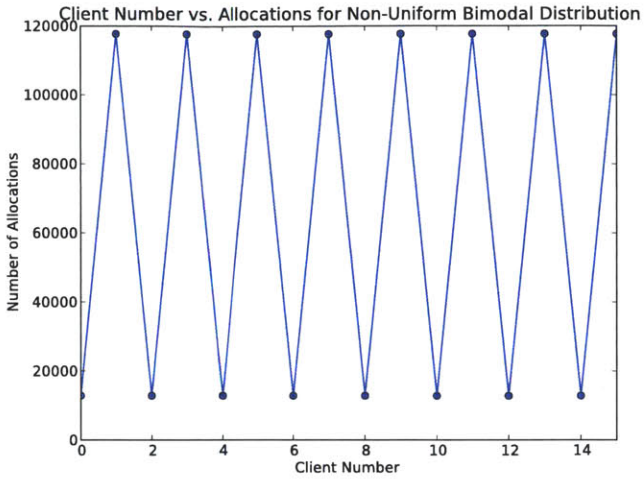

Figure 6-3: Number of allocations completed by each client in the non-uniform, bimodal distribution with 16 clients and 16 servers.

and vice versa. The second phase of this test case reverses the clients from the first phase with every other client either allocating 20% of the nominal pages $pages = (totalPages/numClients) * (0.2) - 256$ or 180% of the nominal pages: $pages = (totalPages/numClients) * (1.8) - 256$. This is shown graphically in Figure 6-4 for 16 clients and 16 servers. We will refer to this distribution as the Non-Uniform Bimodal Two-Phase distribution.

6.2.1 Testing Methodology

Each test is run on an unloaded computer as described above. The PMA servers and test harness clients execute on separate cores. We test 1, 2, 4, 8, and 16 servers and from 1 to 24 clients. Six CPUs are used for other fos system services. Timing is recorded by utilizing the high-resolution, hardware time stamp counter (TSC). At the end of a run, the first client waits for all of the other clients to complete, signaled by a message from every other client. The act of all of the clients contacting the first client at the end of a run serves as a barrier. After the barrier, the first client captures the end of run timing. The metric used throughout this results section is page allocations per million cycles. This is computed by taking the total number of allocations done on every client and dividing it by the number of cycles the worse

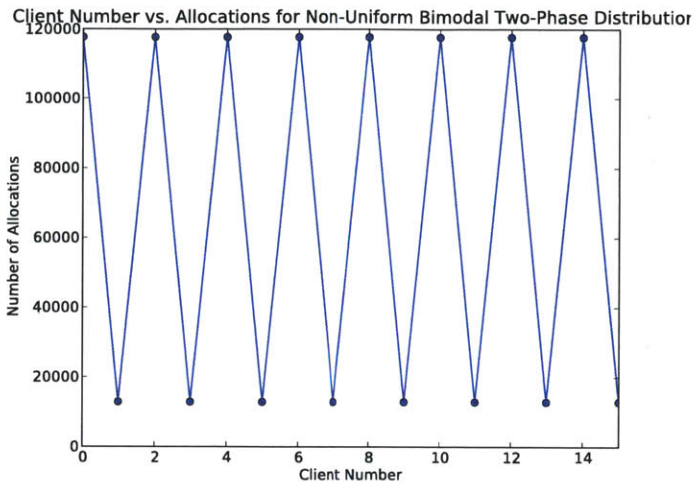

Figure 6-4: Number of allocations completed by each client in the second phase of the non-uniform, bimodal two-phase distribution with 16 clients and 16 servers.

case client took to complete. We use allocations per million cycles in order to make the graphs have non-fractional axes. Figure 6-5 shows an example test configuration with four servers and eight clients.

The graphs in the following section have many data points per line and multiple lines per graph, therefore we take a moment to describe the basic graph structure in the hope of easing graph readability for the reader. Figure 6-6 shows an idealized example graph similar to those used in the remainder of this results section, which we will use to describe the structure of the graphs. First, lets begin with the axis. The vertical axis is the average rate of allocation, mapping, and zeroing of pages per one million cycles. This is an aggregate rate across all of the clients, and a higher point is better. In example, if two clients are used and if there is perfect scaling, the allocations per million cycles will be doubled. The horizontal axis contains the number of clients being used for a particular test. This is just the number of clients, as the number of servers servicing those clients is an independent variable. For reference results, such as Linux, there is no notion of servers, therefore only the number of clients is varied.

Now we direct attention to the legend of the graph, shown in the bottom center. The legend shows the number of servers utilized for each test. Each line in the graph

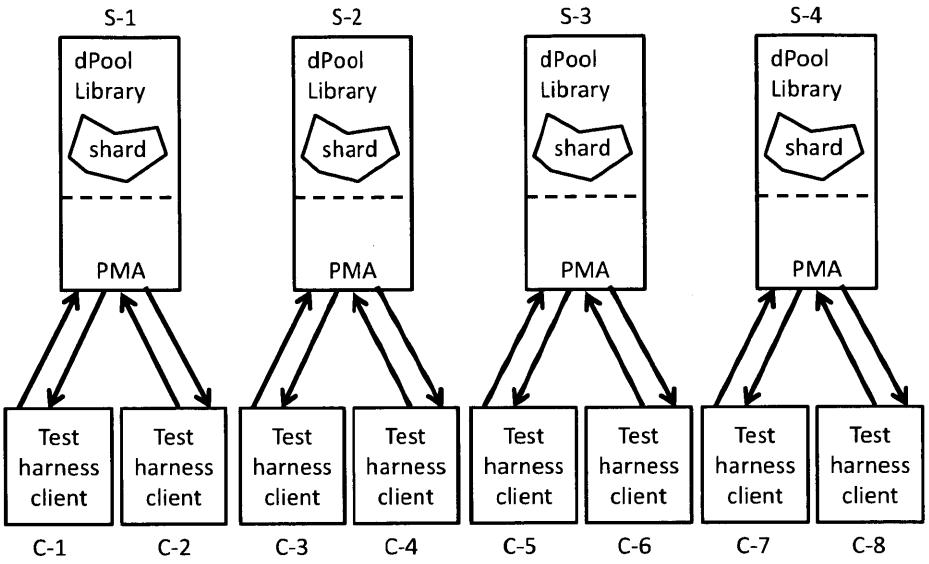

Figure 6-5: An example configuration with four servers and eight clients. The servers and clients are numbered S1-S4 and C1-C8.

represents a different number of servers and each data point represents the number of allocations per millions cycles using a particular number of clients and servers. The title of the graph contains a short description of the algorithm used and the distribution of work utilized for the results in the graph.

We briefly look at different scaling trends as plotted on the example graph. The line with downward-pointing triangles (green) corresponds to two servers. The performance of this configuration shows good scaling, as a function of clients, until approximately nine clients. With greater than nine clients the performance plateaus. The line with right-facing triangles (purple) represents a linear-scaling improvement across all the clients. The slope of the different lines before they plateau is also interesting, as it determines how well the configuration is scaling. A line with a steeper slope scales better than one with a shallower slope. Finally, we see that the line with a box marker (yellow) represents reference data being compared against. In this case, as the legend suggests, the reference is Linux.

For ease of comparison, the graphs in the following section are all graphed with the same axis. While more detail could be seen if the axis were recalibrated per graph, we compare many of the charts with each other in the discussion section, and this is

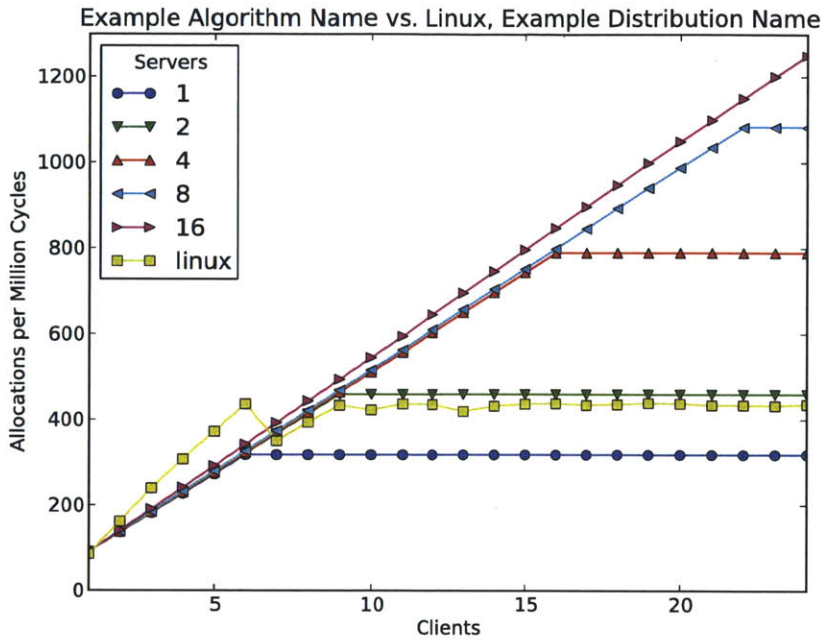

Figure 6-6: Example Results Graph.

made easier if all of the charts are on the same scale.

6.2.2 Reference Comparison

The rest of this chapter evaluates the scalability of different implementations of dPool. The test harness simulates applications which are allocating memory as quickly as possible. As a reference point, for the uniform distribution, a similar workload is executed on a Linux DomU, executing on the same machine and Xen hypervisor. The Linux DomU is Linux version 2.6.28. It is difficult to make an apples-to-apples comparison between two different operating systems due to different code maturity and different functionality sets. Therefore, we give the performance of Linux only as a reference and stress that it is only a reference. Much effort went into making sure that the testing for the fos dPool implementation has similar functionality as the Linux implementation. The Linux implementation allocates pages by touching the first byte in a page as quickly as is possible. This causes the Linux kernel to allocate a physical page, map the page, and zero the page. The test then frees the

page back to the system. The test times how long it takes to execute on different numbers of processors, and the aggregate allocations per million cycles is computed. We only provide the reference point for the uniform distribution as we felt it was overly challenging to have a fair comparison for the non-uniform allocation cases.

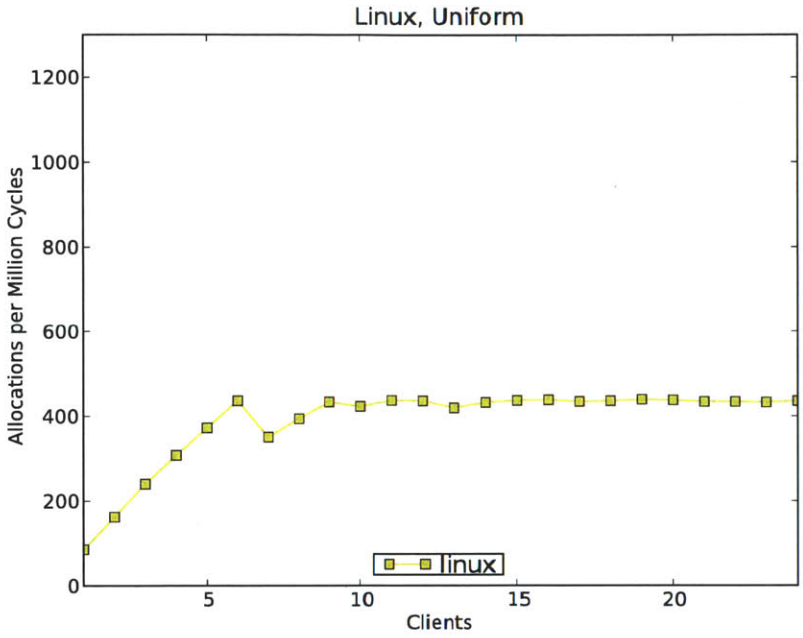

Figure 6-7: Linux Scalability for Uniform Page Allocation Benchmark.

Figure 6-7 shows the scaling of Linux on the uniform distribution test case. For one to six processors, the performance scaling of Linux looks quite good and peaks at 436 allocations per million cycles. After that peak, the performance regresses until it plateaus out around the peak performance. It is interesting to look inside Linux and see why the performance stops increasing. Linux manages free physical memory with a centralized buddy allocator. Linux has a buddy allocator per NUMA node in the system. There are also separate pools for different classes of memory, which are called zones. On 64-bit x86 Linux, the zones are less important as the purpose of the zones is to deal with low-memory, high-memory, and DMA memory, but in 64-bit machines, there is no notion of low or high memory.

Each CPU has a cache of free pages per node. Linux pulls a large number of pages from the central buddy allocator when a local CPU's cache runs dry. If the CPU cache

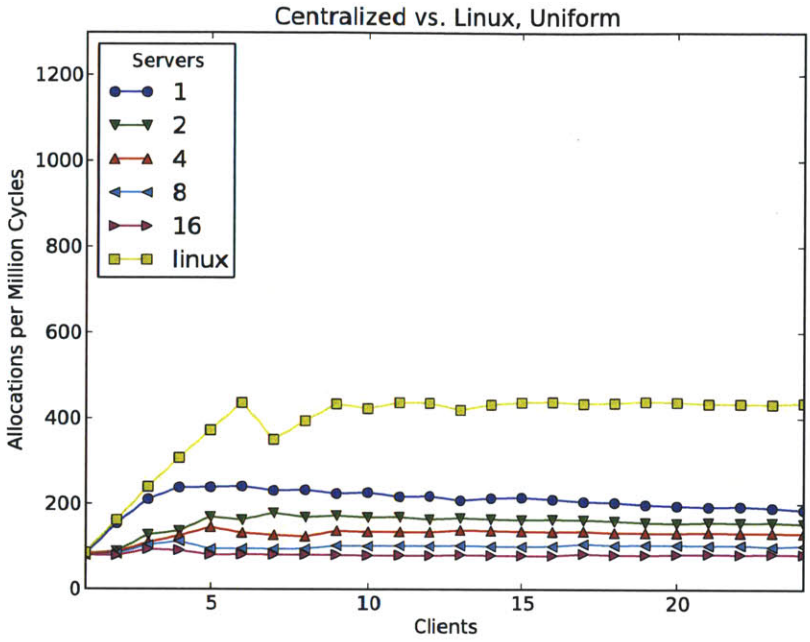

Figure 6-8: Centralized Storage dPool tested with a uniform load compared to Linux.

becomes too full, the pages will be released back to the central buddy allocator. The central buddy allocator has a single lock, named `pg_data_t->zone->lock` protecting access to it. This lock ultimately limits the performance in this test, as many pages need to be retrieved from the central buddy allocator. This was discovered by using `oprofile` as is shown in Figure 3-1. While we tested Linux 2.6.28, the physical page allocation code has not been modified in the newest release of Linux.

6.3 Evaluation of dPool Implementations

The following subsections evaluate the scaling of different dPool algorithms and implementations when being tested with the above workloads.

6.3.1 Centralized Storage

We begin by looking at a dPool implementation which uses centralized storage as described in Section 4.4.1. In this implementation, only one server is capable of storing free pages, while the other servers who use the same dPool interface need to

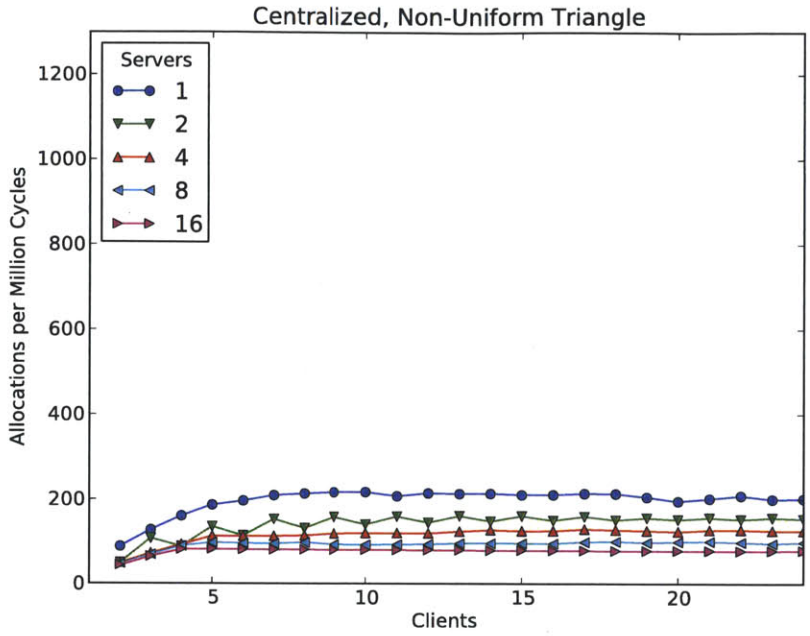

Figure 6-9: Centralized Storage dPool tested with a non-uniform, triangular load.

contact the first server to request or relinquish a page. Figures 6-8, 6-9, 6-10, and 6-11 show the performance trends for the different test workloads.

As expected, centralizing all allocations onto one core quickly becomes a bottleneck and maxes out at four clients in the uniform case and eight clients in the non-uniform cases, after which the performance of this approach decreases. The one server case does the best in all cases. For the test cases which have more than one server, in order for the test harness to allocate a page, it needs to first communicate with its local PMA server. If the local server is not the first server, the dPool contained within the local server sends a message to the first server, which contains the centralized pool storage. This, in effect, increases the communication cost associated with a page allocation, thereby reducing performance as shown in Figure 6-12. It is interesting to see is how the doubling of communication cost along with the way that we benchmark performance affects the results for higher server counts. Clients are evenly distributed between servers. Therefore, as more servers are added, a larger fraction of the work is done by servers which have strictly higher communication costs. For instance, in the one server case, all of the communication goes directly

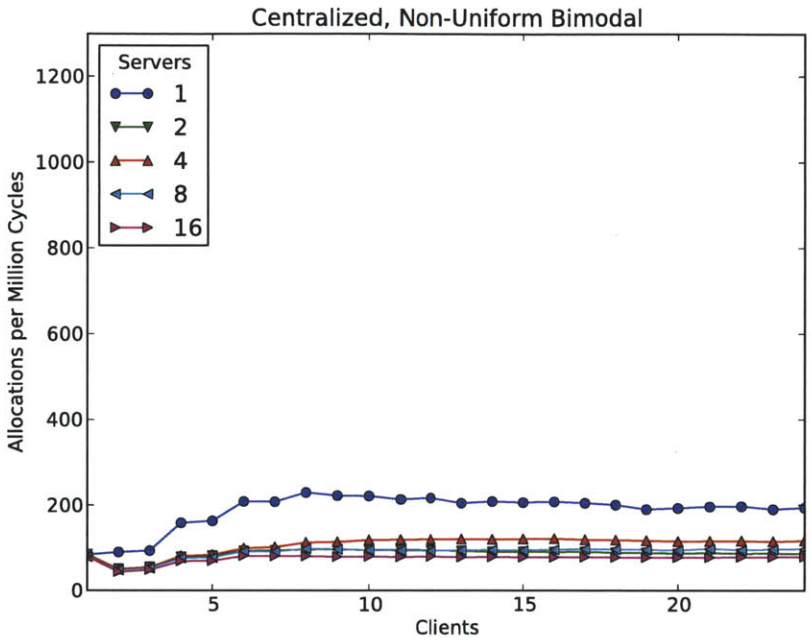

Figure 6-10: Centralized Storage dPool tested with a non-uniform, bimodal load.

between the client and server. For the two server case, half of the workload has a communication cost of one and the other half has a communication cost of two when normalized. For sixteen servers, 1/16 of the work has a communication cost of one and 15/16 of the work has a communication cost of two. Therefore, as more servers are added, a larger percentage of the work done has a strictly higher communication cost and therefore lower performance. It should be noted that the centralized storage implementation does worse than the reference Linux implementation at all client numbers.

In figure 6-9, a well-defined, alternating ripple can be seen in the two server case (green line with downward facing triangle markers). This is because, with an odd number of clients, one more client maps onto the centralized server, thereby enabling higher performance.

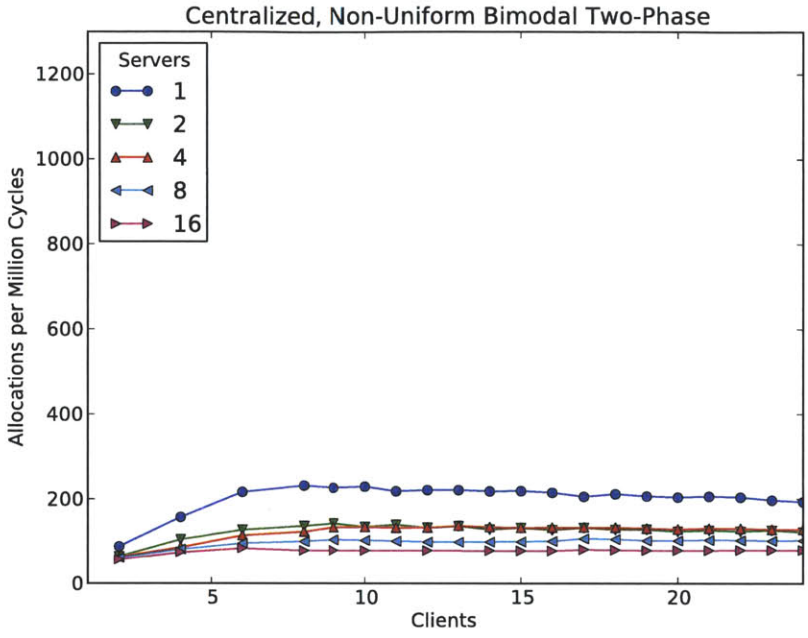

Figure 6-11: Centralized Storage dPool tested with a non-uniform, bimodal two-phase load.

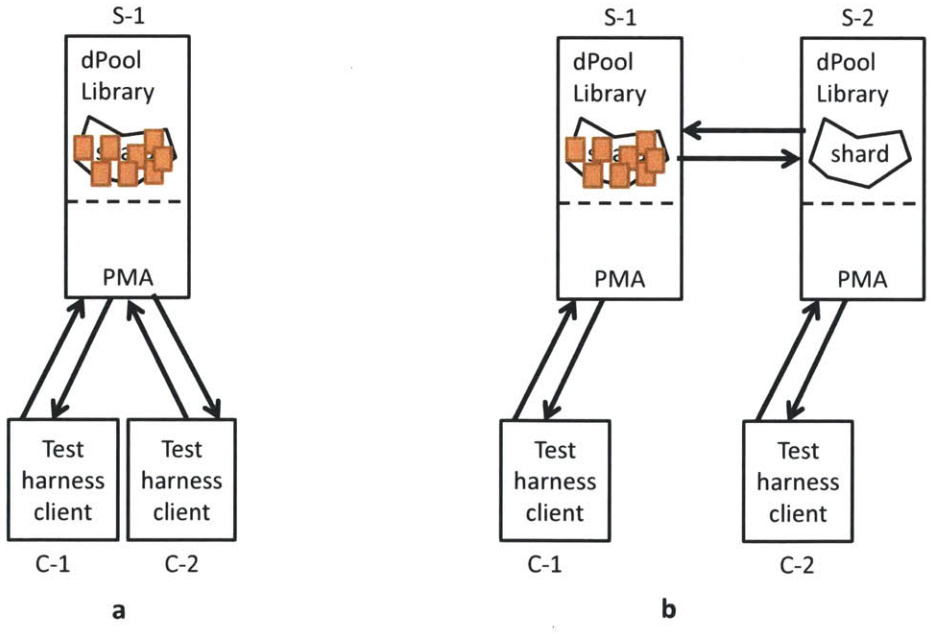

Figure 6-12: a) Centralized dPool with one server and two clients. Both clients directly communicate with a server which can contain elements. b) Centralized dPool with two servers and two clients. Only the first client can directly communicate with the server which contains elements in its dPool. The second client must incur higher communication cost as it needs to communicate with server S-2 which in turn needs to message server S-1 in order to fulfill any requests for dPool elements.

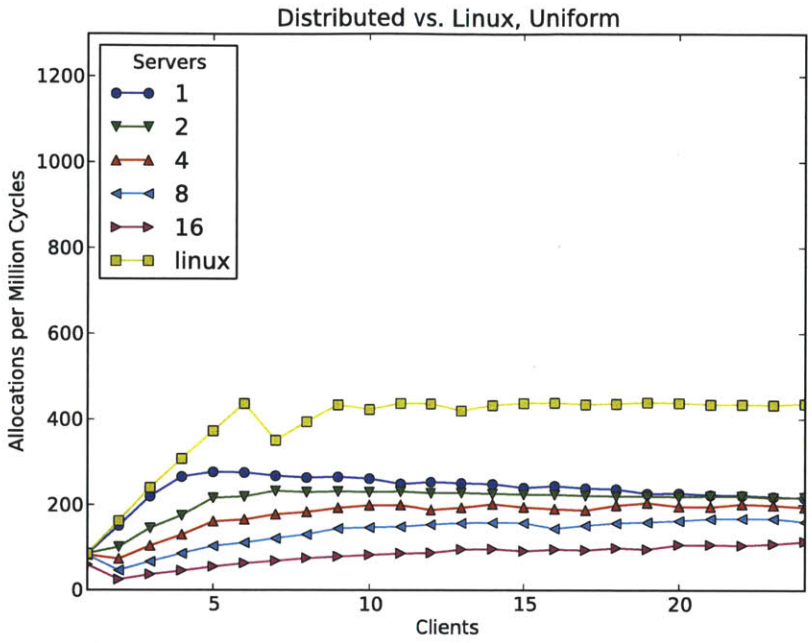

Figure 6-13: Distributed Storage dPool tested with a uniform load compared to Linux.

6.3.2 Distributed Storage

This section shows the performance for a dPool that allows pages to be distributed across multiple servers' storage as described in Section 4.4.2. Figures 6-13, 6-14, 6-15, and 6-16 show the performance trends for the different test workloads.

Like in the centralized storage implementation, the distributed storage case quickly maxes out on performance. For the uniform case, it maxes out after five clients, while the non-uniform cases fare slightly better, with the non-uniform, triangle case even showing some scaling for the multi-server cases. What is interesting to see is that this algorithm actually demonstrates anti-scalability as more servers are added. This is due to pages being allocated initially on the first server in the PMA fleet. For the same reasons described for the centralized case, if the PMA server only pulls one entry at a time, it reverts to the centralized storage case. If the test case contained more mid-test freeing and reuse of pages on the same server, we would expect the distributed algorithm to do much better in the long run. The non-uniform, bimodal two-phase workload contains some page reuse, but unfortunately still transfers many

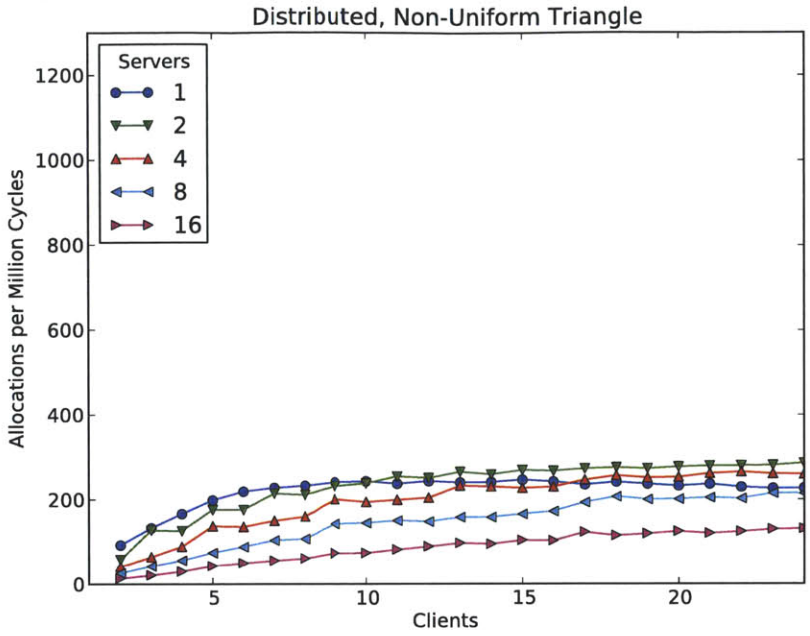

Figure 6-14: Distributed Storage dPool tested with a non-uniform, triangular load.

pages between servers because the distribution reverses which servers are loaded between the two phases.

It is interesting to note that for higher numbers of servers and low number of clients, the distributed algorithm actually does worse than the centralized case as can be seen when comparing Figure 6-8 and Figure 6-13 at the two-client point. Investigating this anomaly reveals one of the largest differences between these two strategies. In the distributed storage case, when a server runs out of free pages, it contacts another server. It does this in a round-robin fashion, while in the centralized storage case, it contacts the server which is guaranteed to have a page. Therefore, in the distributed storage case, as the number of servers increases there are more locations to check for free pages. Because all of the pages start on the first server, in effect this causes the server to check many other servers which are guaranteed not to have a page before contacting the first server.

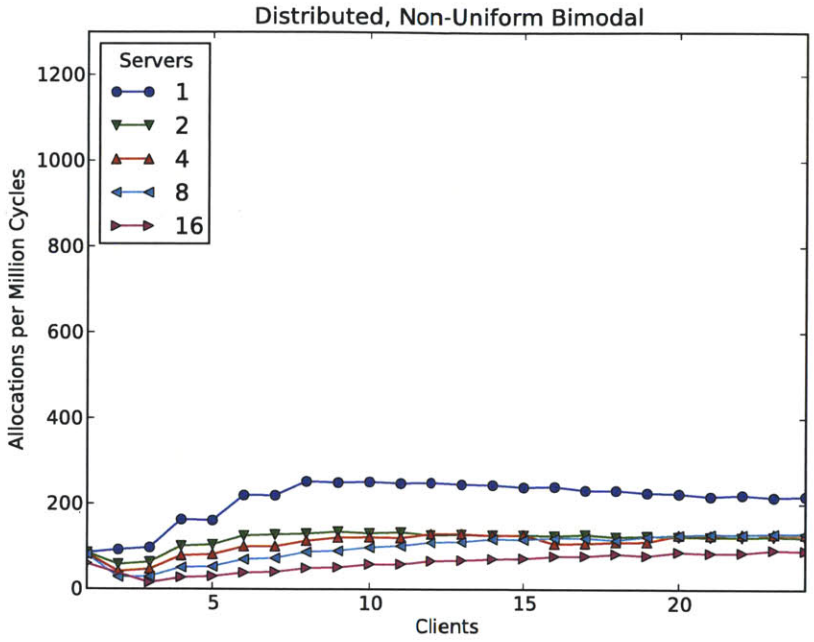

Figure 6-15: Distributed Storage dPool tested with a non-uniform, bimodal load.

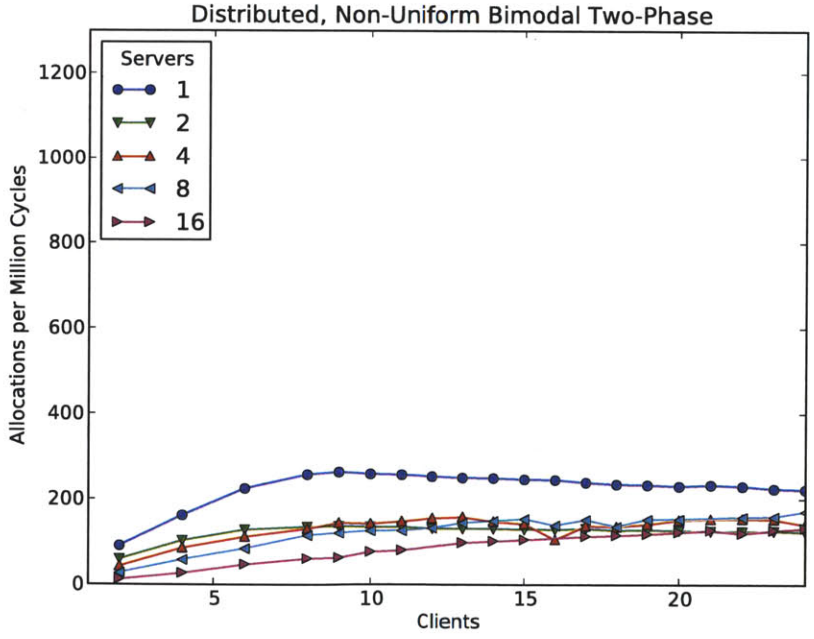

Figure 6-16: Distributed Storage dPool tested with a non-uniform, bimodal two-phase load.

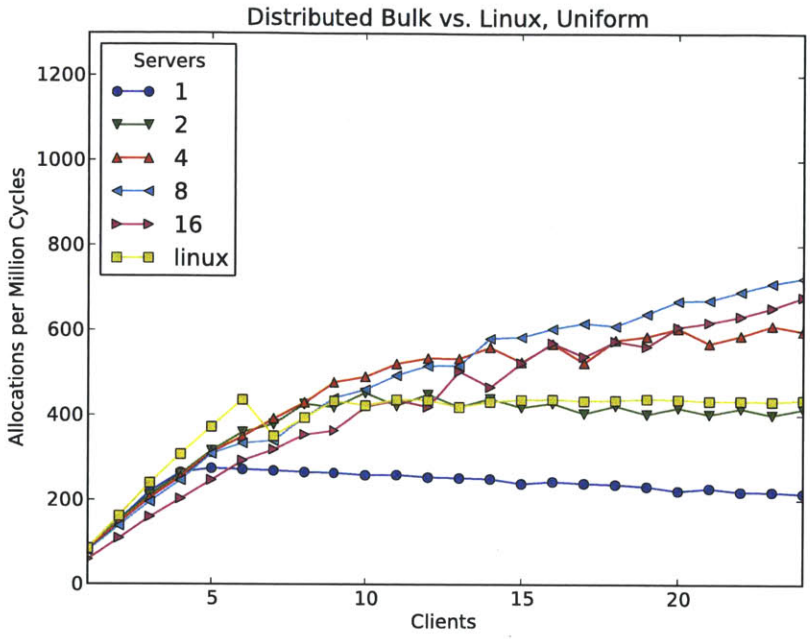

Figure 6-17: Distributed Storage Bulk Transfer dPool tested with a uniform load compared to Linux.

6.3.3 Distributed Storage Bulk Transfer

Figures 6-17, 6-18, 6-19, and 6-20 show the results for a distributed storage dPool which transfers up to 50 pages in one message as described in Section 4.4.3. This strategy has several advantages. First, it reduces the messaging cost to move pages by a factor of 50. Second, it reduces the probability that a server which is being contacted by another server contains no free pages, as most servers will have some level of free pages stored within their dPool.

Looking at the uniform distribution, Figure 6-17, we can see that four or more servers can provide better performance than the reference Linux implementation. Also, as more clients are added, the performance trends up, showing that this implementation has some scalability in terms of adding more clients. This graph also shows that the number of servers begins to become a bottleneck and ultimately limits scalability, as can be seen with the one, two, and four server cases. Another interesting occurrence is that the 16 server case shows good scalability when adding clients, but performs worse than the eight server case. When there are fewer than 16 clients,

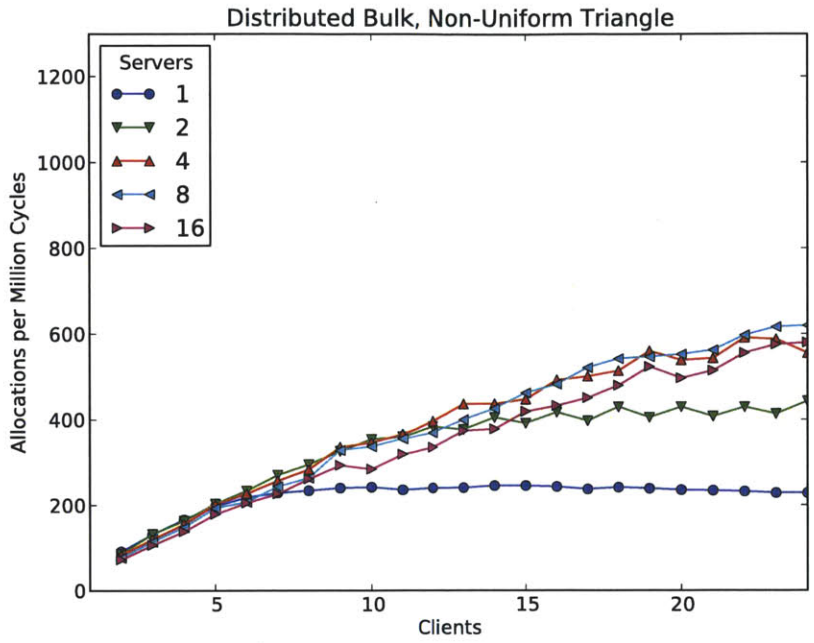

Figure 6-18: Distributed Storage Bulk Transfer dPool tested with a non-uniform, triangular load.

the average communication cost for 16 servers ends up being higher than the eight server case. This is because, as described above, a server which runs dry of pages communicates with other servers in a round-robin fashion. If the server being queried for pages is one of the servers which is not servicing a client, it acts as dead weight. Although the client-less server gets queried, it will never have pages to share, thereby increasing the communication cost when compared to the eight server case.

Above 16 clients, the trends are interesting. As can be seen in both the 16 server and eight server case, performance continues to improve as more clients are added. In the uniform load case, 16 servers performs slightly worse than the eight server case. This is largely due to a load imbalance which occurs on the servers. In our test, one client only communicates with one server in order to preserve communication locality, as is the philosophy of fos. This causes the load, when there are fewer servers, to be more evenly distributed. For instance let us compare when there are 20 clients communicating with eight servers versus 20 clients communicating with 16 servers. With 20 clients and eight servers, four of the servers are servicing three

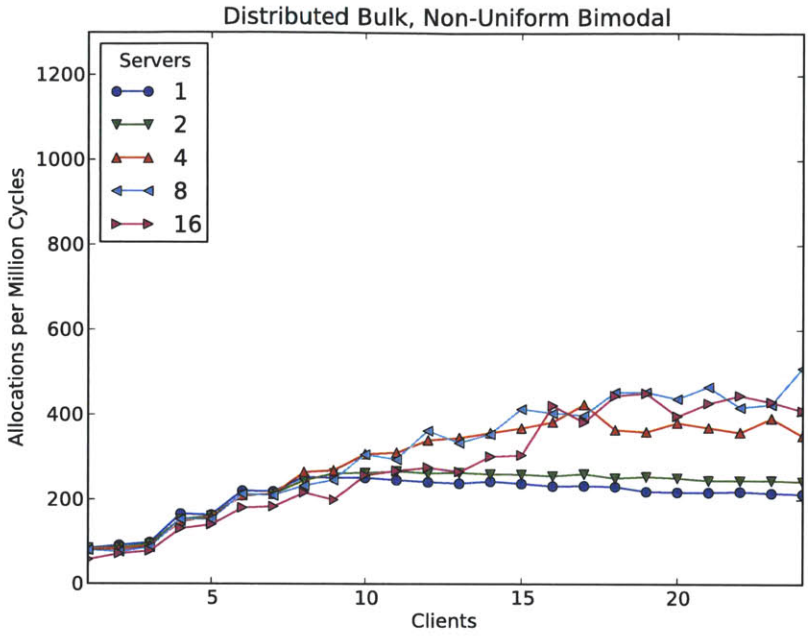

Figure 6-19: Distributed Storage Bulk Transfer dPool tested with a non-uniform, bimodal load.

clients and four servers are servicing two clients. For the 16 server case, four of the servers are servicing two clients and twelve of the servers are servicing one client. So, if we compare the load of the least loaded server to the maximum loaded server, we see that in the eight server case, each server has either 3/3 or 2/3 of the maximum load and the load difference is only 1/3, while in the 16 server case, each server has either 2/2 or 1/2 of the maximum load. Therefore, the difference in terms of maximum load is larger with more servers. Not only is the load difference greater, but also, the variation across servers. For instance, in the eight server case, 4 out of 8 servers are lightly loaded, while in the 16 server case, 12 out of 16 servers are lightly loaded. This impacts performance because we measure the time taken by the least performing client/server pair. If we were to distribute requests from clients to servers more evenly, we should expect some of this small performance gap to be reclaimed, but that goes against the locality philosophy of fos.

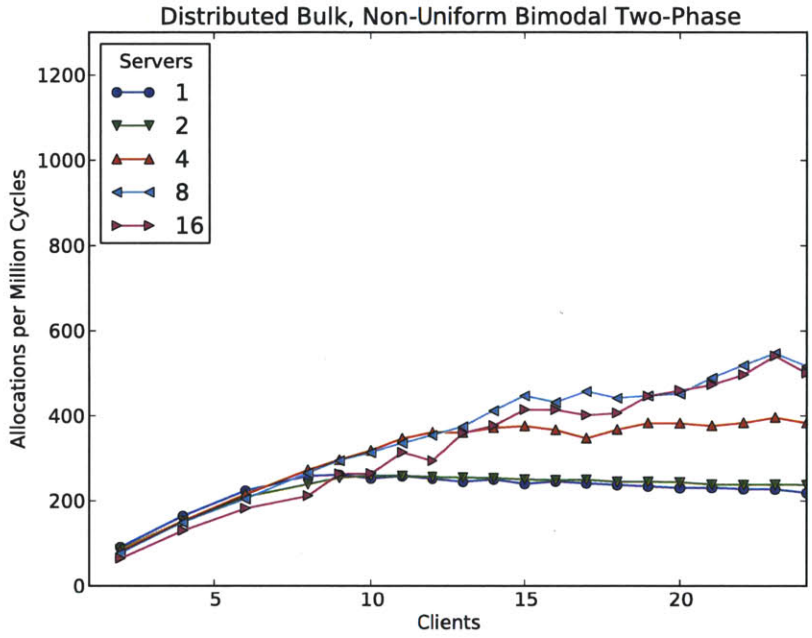

Figure 6-20: Distributed Storage Bulk Transfer dPool tested with a non-uniform, bimodal two-phase load.

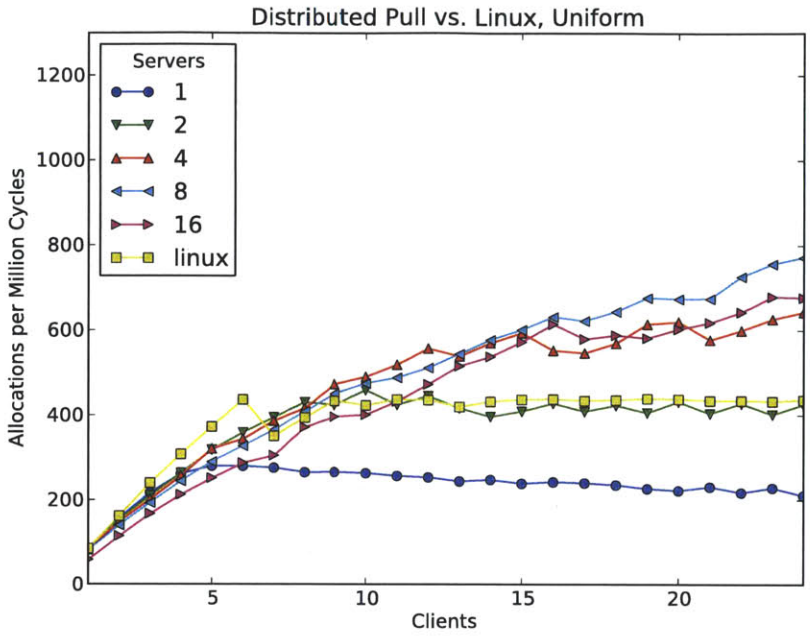

Figure 6-21: Distributed Storage Bulk Transfer with Background Pull dPool tested with a uniform load compared to Linux.

6.3.4 Distributed Storage Bulk Transfer with Background Pull

Figures 6-21, 6-22, 6-23, and 6-24 show the results when we add a thread which pulls elements between dPool *shard*s in the background. When comparing this to the previous case, there is little to no performance improvement and the results look very similar. This is not too surprising, as a pull protocol requires the loaded server to request free elements. Also, preemptive pulling does not decrease the number of page pull requests. It even makes matters worse when the number of clients is less than the number of servers, as it will horde pages by pulling pages into servers which will never be directly contacted by a client and allocated from.

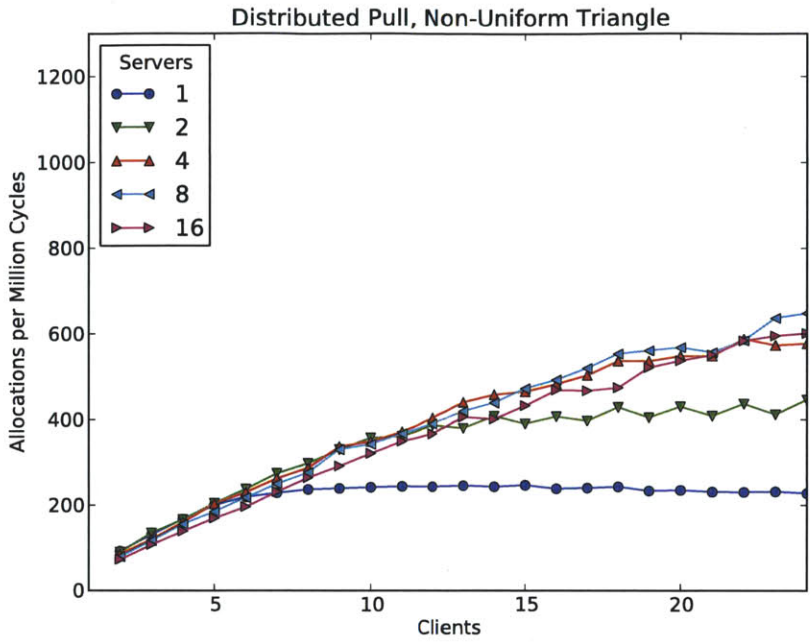

Figure 6-22: Distributed Storage Bulk Transfer with Background Pull dPool tested with a non-uniform, triangular load.

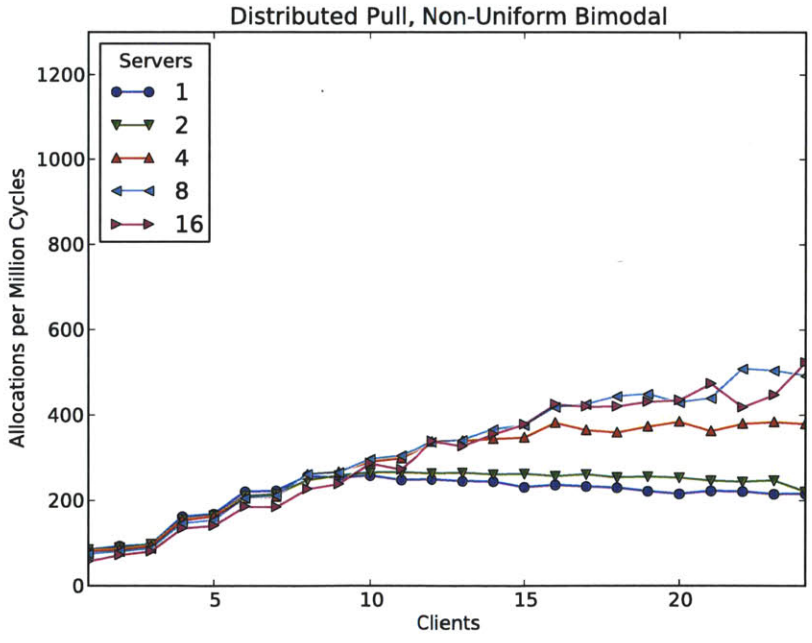

Figure 6-23: Distributed Storage Bulk Transfer with Background Pull dPool tested with a non-uniform, bimodal load.

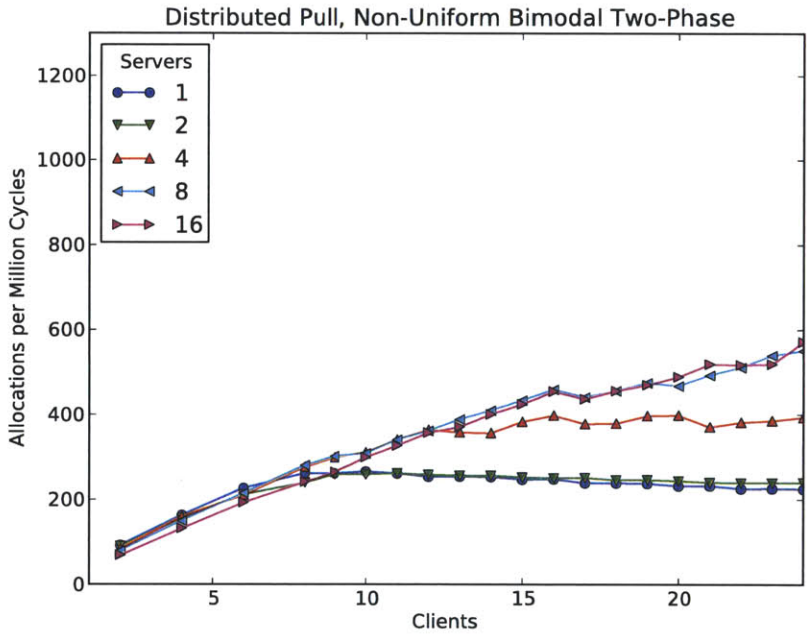

Figure 6-24: Distributed Storage Bulk Transfer with Background Pull dPool tested with a non-uniform, bimodal two-phase load.

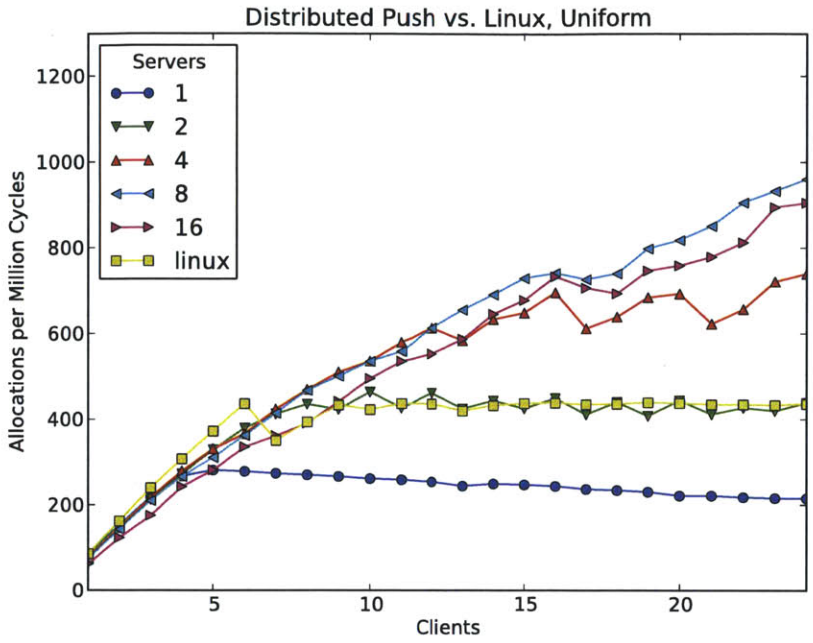

Figure 6-25: Distributed Storage Bulk Transfer with Background Push dPool tested with a uniform load compared to Linux.

6.3.5 Distributed Storage Bulk Transfer with Background Push

In this section we replace background threads which pull elements with background threads that push elements from dPool *shard*s which have many elements to other *shard*s. Figures 6-25, 6-26, 6-27, and 6-28 show the results for the bulk transfer with background push algorithm described in Section 4.4.5. A similar trend appears as in the previous graphs where the PMA fleet is able to increase performance by adding more clients. Also, in general, the one, two, and four server cases show signs of plateauing performance, indicating that they ultimately will limit performance. The eight and 16 server trends continue to increase in performance as more clients are added, up to the maximum number of tested clients.

Overall performance is better than the previous algorithms across all workloads. The background push is effective at distributing pages between all of the different dPool *shards*. This is especially useful at the beginning of a test where all of the

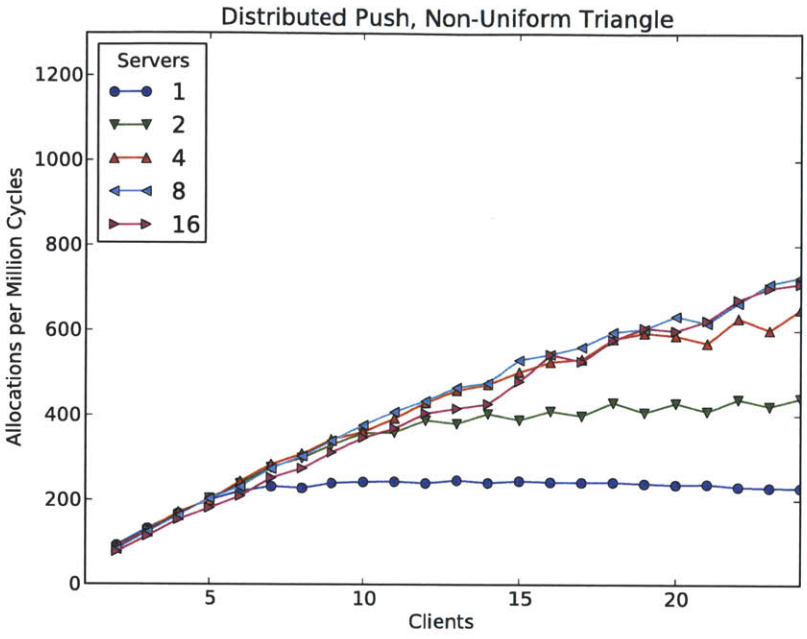

Figure 6-26: Distributed Storage Bulk Transfer with Background Push dPool tested with a non-uniform, triangular load.

pages begin on the first server.

Load imbalance continues to haunt the 16 server case when compared with the eight server case for large numbers of clients. This causes a new phenomenon for low number of servers. Because the total number of pages in the page pool are fixed and limited, the push algorithm actually pushes pages to dPool *shards* with no clients attached. Therefore these servers accumulate pages until they ultimately reach the push threshold where they will start pushing elements to other servers. Because the push threshold is relatively high, a good number of pages are taken away from the larger pool and stored in the non-useful, clientless, servers.

One interesting feature to note about this algorithm is that it indiscriminately pushes pages. This can cause a page to be transferred multiple times as servers will push pages to servers which have a copious number of pages. Therefore, we look toward more intelligent algorithms.

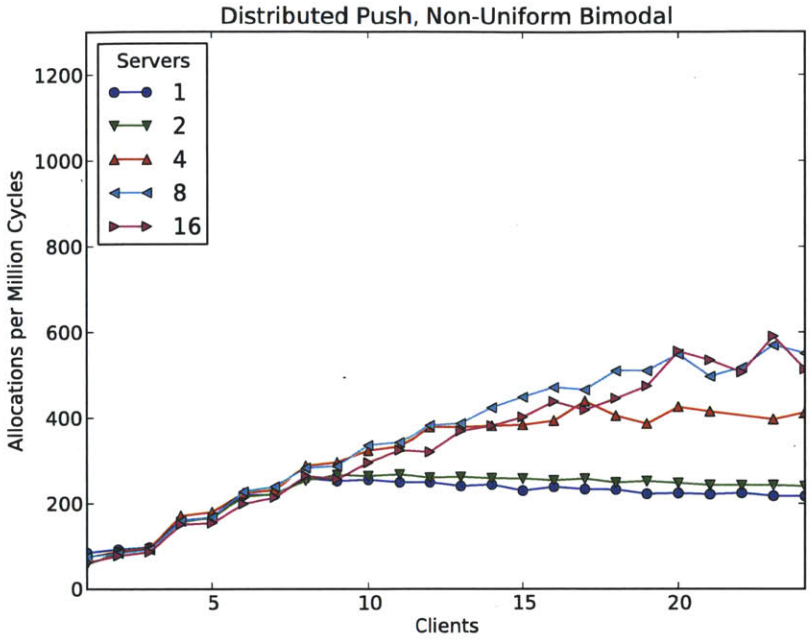

Figure 6-27: Distributed Storage Bulk Transfer with Background Push dPool tested with a non-uniform, bimodal load.

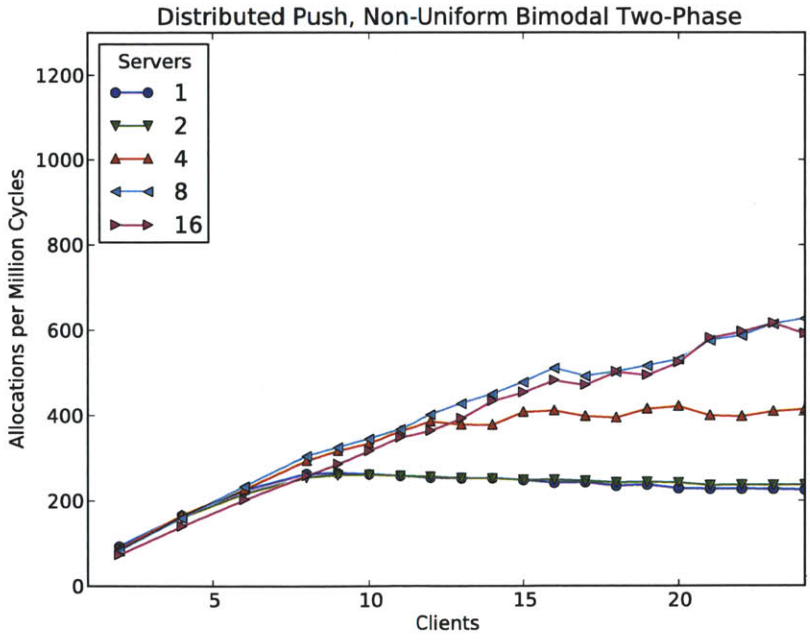

Figure 6-28: Distributed Storage Bulk Transfer with Background Push dPool tested with a non-uniform, bimodal two-phase load.

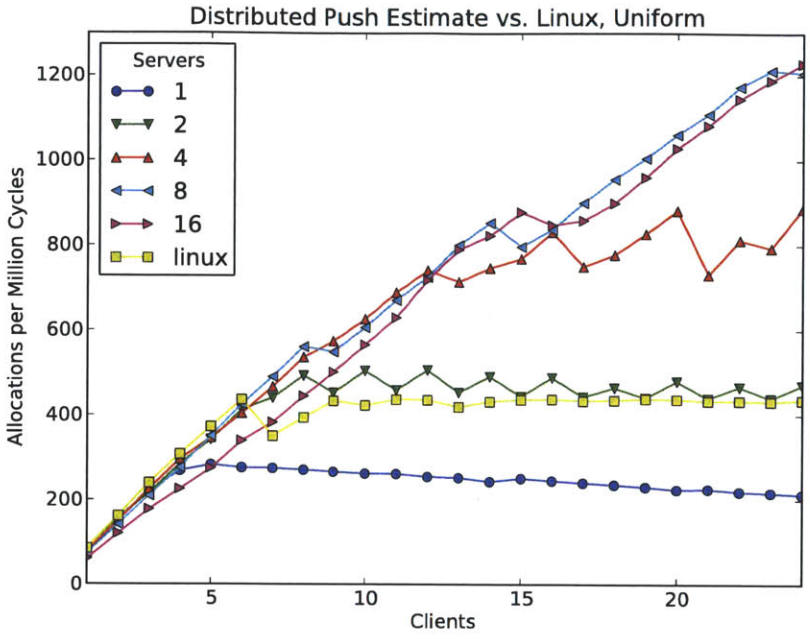

Figure 6-29: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool tested with a uniform load compared to Linux.

6.3.6 Distributed Storage Bulk Transfer with Background Push and Element Estimation

We now explore adding the ability to make intelligent decisions about where to push spare pages. Because dPool operates in a distributed environment, determining which dPool *shard* to push to requires extra messages to be sent and computation to be used. Therefore, the benefit of better knowledge must outweigh the cost of communicating estimate information for this approach to increase performance. The algorithm, as described in Section 4.4.6, uses a background protocol to estimate the number of elements that each dPool *shard* contains. Figures 6-29, 6-30, 6-31, and 6-32 show how well this approach performs on the four workloads.

Looking at the results, we see that for all workloads, the one, two, and four server cases all plateau after reaching a maximum performance. The four server case provides better peak performance than the two server case, which is better than the one server case, thereby showing scalability in the number of servers. The eight and

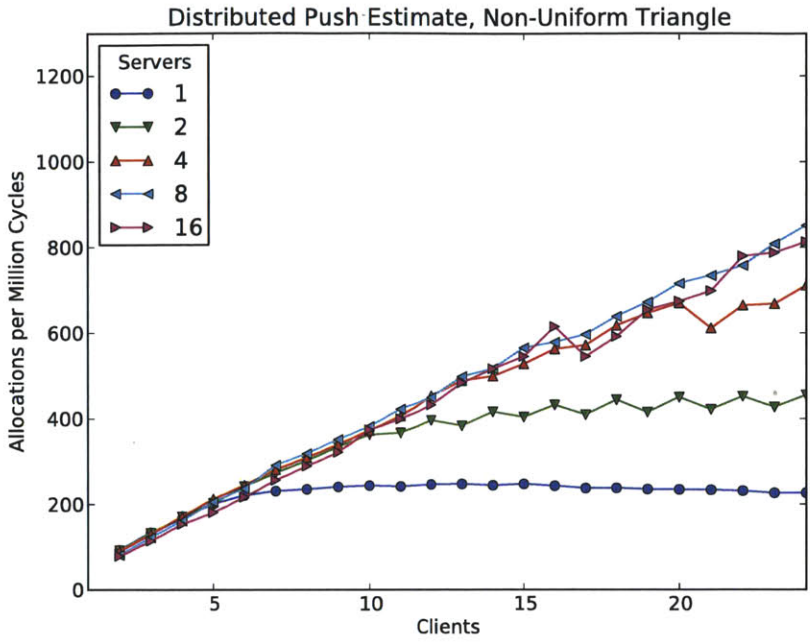

Figure 6-30: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool tested with a non-uniform, triangular load.

16 server cases show good scalability with the number of clients and continue to scale beyond the measured number of clients. The slope of the scalability is close to the slope of and absolute performance of the Linux reference for low numbers of clients. For two or more servers, the fos implementation achieves greater performance than the reference implementation.

We now focus on comparing the 16 server and eight server cases. For fewer than 16 clients, the performance of 16 servers is slightly worse than that of eight servers. We investigated this by enabling other performance metrics in our runs and found that in the eight server case, no demand requests for pages occur, because the push mechanism is effective at delivering pages to servers before they are needed, thereby saving critical path communication cost. In the 16 server case, the push mechanism pushes pages to servers which do not have any clients attached. Later, when the test gets into a low page regime, the dPool *shard* inside of the loaded servers need to demand request pages from the otherwise idle servers, thereby impacting performance. One possible way to solve this is to use fewer servers than clients as the PMA

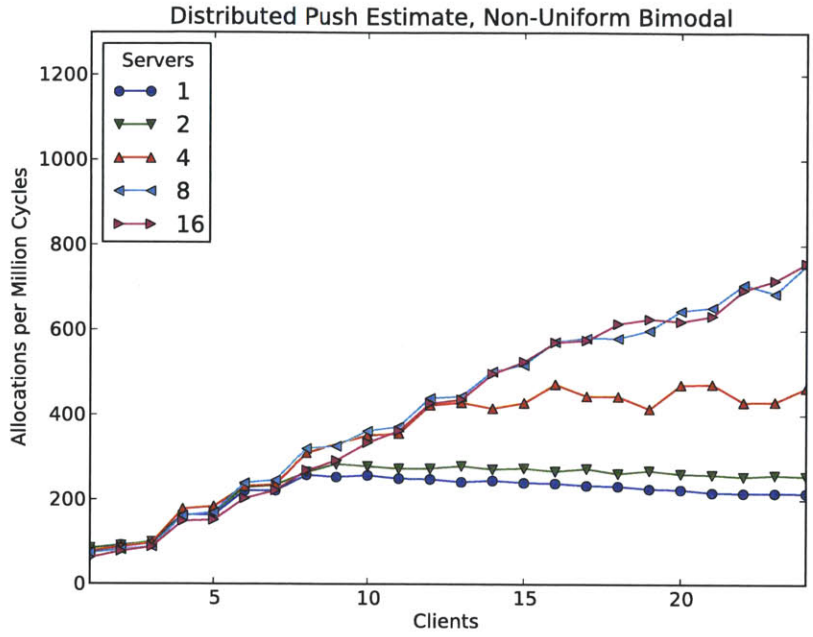

Figure 6-31: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool tested with a non-uniform, bimodal load.

and dPool can elastically change the number of CPUs being used. Another possible solution is to vary the threshold below which dPool *shards* do not push elements. The threshold could be set to vary with the number of elements held within the global dPool *instance*, thereby encouraging idle servers to push elements to servers in need of elements in the low element case.

Above 16 clients, the performance trends for both eight and 16 servers track closely together, but the performance for the eight server implementation is slightly better (less than 5% better). We investigated and found that this performance difference is for a few reasons. First, the load between servers is more balanced with fewer servers. Because our performance metric measures the time required by the slowest client, load balance matters quite a bit. For a fixed number of clients greater than 16 clients, the percentage difference in load as seen by the servers is greater with more servers. This is because we statically assign clients in a round-robin fashion to servers. Therefore, with fewer servers, the clients wrap around the servers more quickly. For instance, assuming 20 clients and eight servers, four servers will be communicating with two

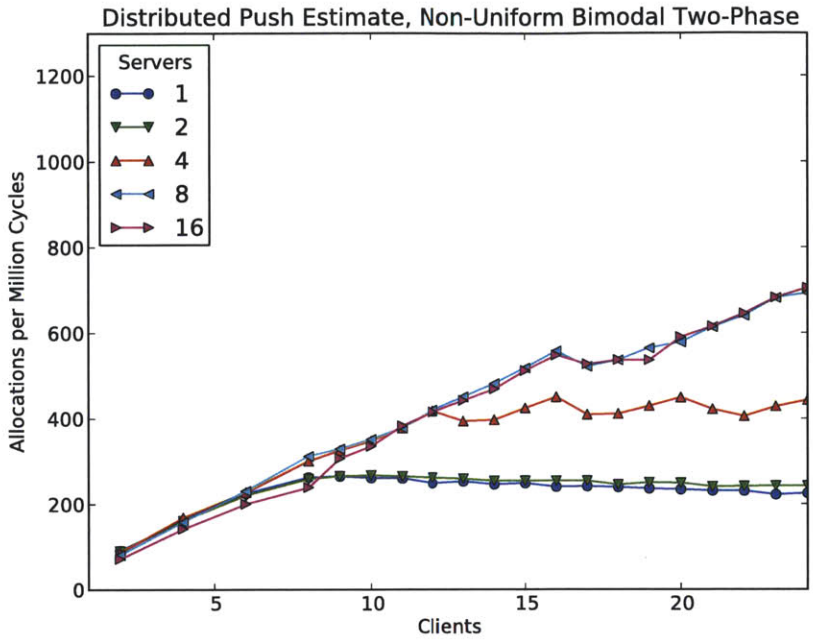

Figure 6-32: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool tested with a non-uniform, bimodal two-phase load.

clients and four will be communicating with three clients. If this same load is run with 16 servers, four servers will be servicing two clients and 14 will be servicing one client. In the eight server case, the load difference is 3/3 vs. 2/3 while in the 16 server case, the load difference is 2/2 vs. 1/2. Therefore, the load difference as a percentage is actually larger in the 16 server case. Also, the number and percentage of servers which are lightly loaded in the 16 server case is quite a bit higher, with 12/16 in the 16 server case versus 4/8 in the eight server case. When we look at the 16 client case, we see that there is perfect load balancing on both the eight and 16 server case as 16 is evenly divisible by both eight and 16. When the load is perfectly balanced, the 16 server case exhibits better performance than the eight server case across the first three distributions.

Unfortunately, some portions of the algorithm become more expensive as the number of servers increases. For instance, searching for a free page is linear with the number of servers. Also, the estimation calculation has to send a quadratic number of messages when an update occurs. However, the estimate calculation occurs

infrequently enough that it does not appear to limit scalability.

Last, in the eight server case, the layout for the clients is slightly better than the 16 server case as the 16 server case pushes everything slightly farther apart. We expect that for some larger number of clients, the performance for eight servers will plateau off and the performance for 16 servers will continue to increase, but we have not been able to reach that number of clients on our current test computer.

In conclusion, we find that the cost of pushing out updates to estimate the number of elements that each dPool *shard* contains has good paybacks and increases the scalability of the dPool as tested.

6.3.7 dPool Algorithm Comparison

By looking across the different algorithms presented and tested, we see that with the better algorithms, it is possible to create a dPool data structure which is able to provide good scalability in terms of adding clients. Also, the maximum performance achievable before performance plateaus scales with the number of servers. Several insights can be gleamed by looking across the results.

First, when comparing Figure 6-17 to Figure 6-8, we can see that adding the bulk distribution of dPool elements over a distributed storage implementation helps performance and scalability.

Second, adding a background thread which pulls elements, comparing Figure 6-21 to Figure 6-17, does not improve performance much. In contrast, adding a background thread which pushes elements from one dPool *shard* to another does significantly improve performance and scalability as shown by comparing Figure 6-25 to Figure 6-17.

Adding the ability for each dPool *shard* to intelligently make decisions about which other *shards* to push elements to significantly improves performance and scalability when compared to pushing in a round-robin fashion, as shown by comparing Figure 6-29 to Figure 6-25. This was not completely obvious as there is cost involved in updating a *shard*'s estimation of the size of the local pool that each other *shard* contains. The overhead of updating this information trades off against the quality of the element count estimation.

One outcome which we were a little bit surprised about is that the better algorithms were better for all of the different workloads. We were expecting that one of the workloads would favor one of the algorithms over the other and that they would not be a strict hierarchy.

One nice feature of fos's fleet design and the design of dPool is that the number of servers and dPool *shards* can be dynamically adjusted in an elastic manner. This, in effect, enables the fos system to follow the highest line in the curves presented in previous figures by adjusting the number of servers for the number of clients and load.

Figure 6-33: a) Eight servers placed close together with clients placed close together versus b) servers distributed on processors close to the clients they serve.

For instance, from a systems perspective, it might be best to utilize a lower number of servers until that number of servers limits performance and only then utilize more CPUs by switching to a larger number of servers and dPools.

6.3.8 Placement

In order to see if the placement of servers and clients effects performance, we compared a placement which places all of the servers on nearby cores versus placing the client near the server which services it. Figure 6-33 shows these two placement configurations. We present the results for the distributed storage bulk transfer with background push and element estimation implementation, but we have found similar results for the distributed storage bulk transfer case. The previously presented Figures 6-29, 6-30, 6-31, and 6-37 show the results when placing the server near the client it is servicing (option b in Figure 6-33). Figures 6-34, 6-35, and 6-36 show the same algorithm with a placement which puts all of the servers together and all of the clients together (option a in Figure 6-33).

When comparing these different placements, we find that placing the server near the clients which it services provides a small performance improvement over co-locating servers. This suggests that communication between servers is less important than communication between a client and its servicing server. This is not unexpected as the algorithms that dPool utilizes work to minimize communication between servers. For example, they do bulk element transfers and rebalancing of elements off of the critical computation path. These results also show that the AMD machine that these results were gathered on does not have uniform communication costs, but that the non-uniformity is modest.

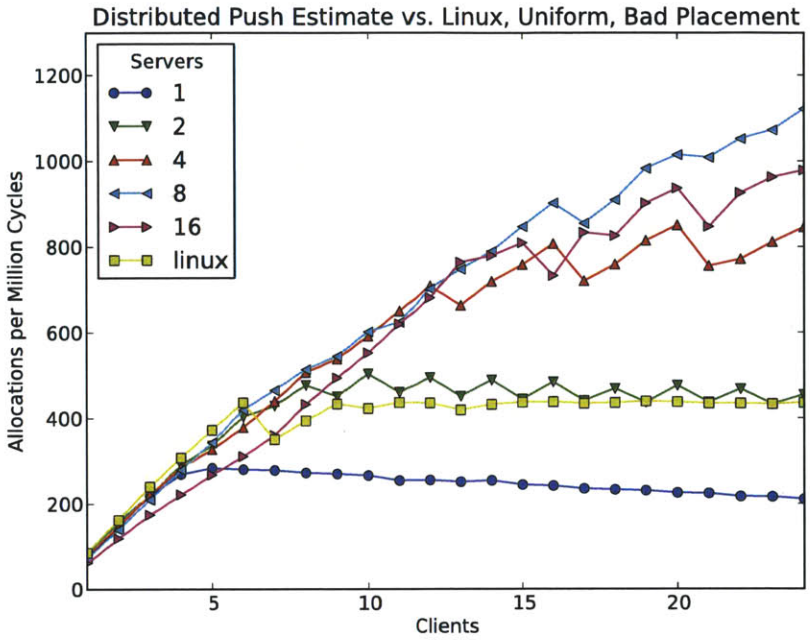

Figure 6-34: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool with poor placement and tested with a uniform load compared to Linux.

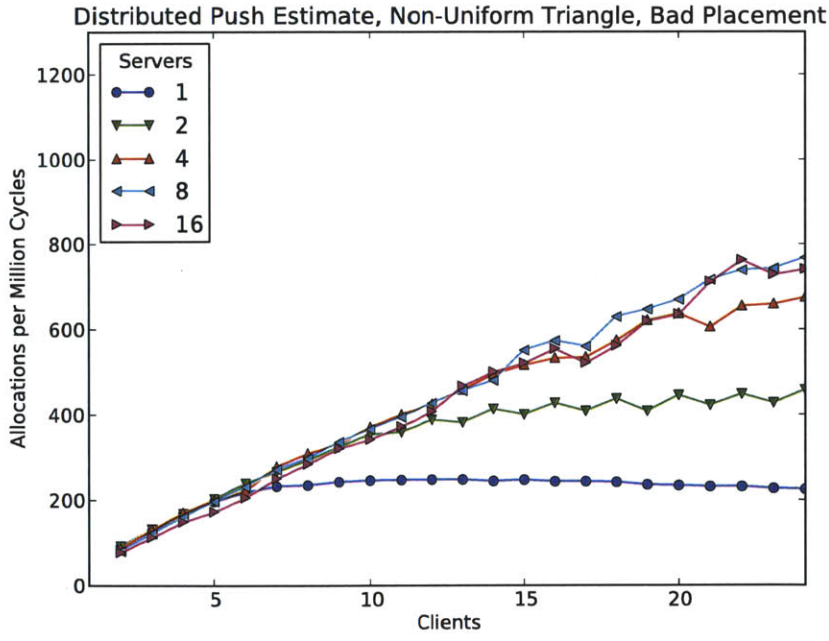

Figure 6-35: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool with poor placement and tested with a non-uniform, triangular load.

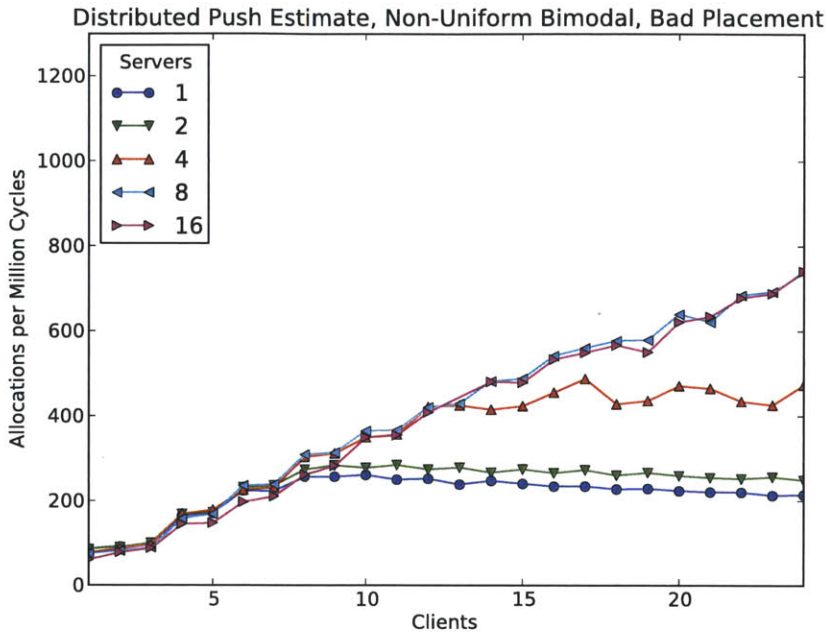

Figure 6-36: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool with poor placement and tested with a non-uniform, bimodal load.

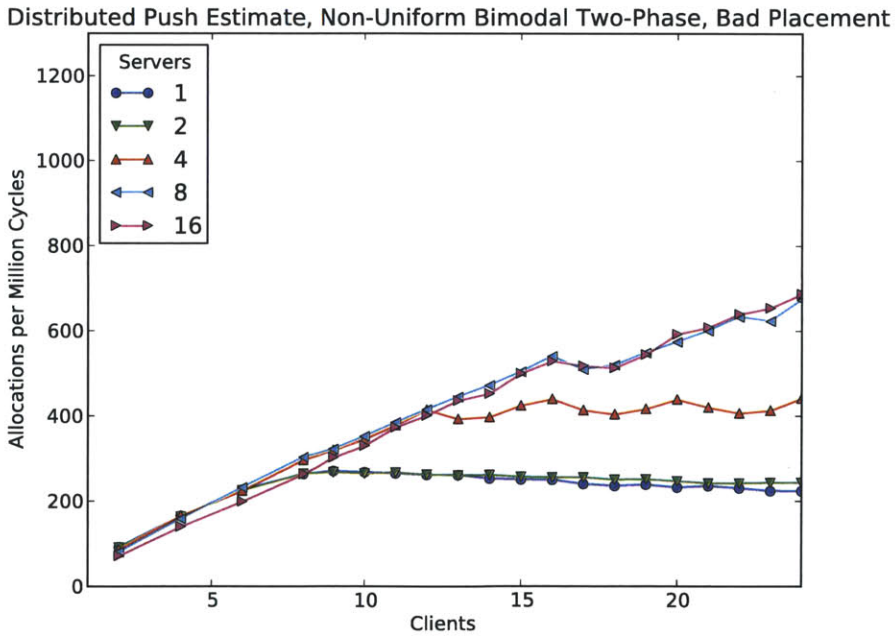

Figure 6-37: Distributed Storage Bulk Transfer with Background Push and Element Estimation dPool with poor placement and tested with a non-uniform, bimodal two-phase load.

Chapter 7

Conclusions

Through the process of creating the dPool distributed data structure, we have gained several insights. The foremost insight is that a message passing based, low-level OS service, such as the fos Physical Memory Allocation service can be successfully split into a dPool distributed data structure and the service main functionality while providing good performance and scalability. In Chapter 6, we showed that dPool empirically provides scalable performance as the number of clients using the Physical Memory Allocation server fleet increases across a set even and uneven micro-benchmark workloads. We also showed that the maximum performance deliverable by dPool before performance plateaus scales with the number of dPool *shards*.

In Chapter 4, we described the construction of the dPool and how parallel and distributed programming techniques can be applied to its construction. We showed elements being partitioned across multiple dPool *shards*. We also described dPool utilizing lazy estimation of the number of elements that each dPool *shard* contained as a way to make intelligent decisions while not needing exact information. Finally, we described how dPools can grow and shrink in size in response to load.

In Chapter 6, we explored different algorithms being used inside of dPool and uncovered insights. First, dPools benefit from having background threads. As is shown for both background pulling and background pushing, adding threads that operate during idle time can effectively rebalance load. Second, we found that pushing elements in the background to be superior to pulling elements in the background.

We found that this is because pushing elements occurs on cores which are otherwise unloaded and these cycles are truly spare. In the pulling case, cores which are already on the critical path do not have much to gain from pulling preemptively versus simply pulling when they run out of elements as the communication occupancy is the same. Empirically, we found that pushing elements from *shard*s which contain large numbers of elements to *shard*s with fewer is superior to pushing elements indiscriminately. One interesting insight is that the cost of keeping element list size information consistent is outweighed by the performance gain of using list size information.

Finally, some other insights include that the placement of fos servers near the clients they serve is important for performance even on relatively uniform machines such as the one we tested on. This will likely become more important on future multicore processors where the communication latencies will become larger and less uniform. Also, the fleet approach allows the delivered fleet performance to ride the maximal envelope of performance provided by different numbers of dPool *shard*s and fos fleet servers.

7.1 Future Directions

In the future, we expect that the fos project will continue to grow and need further distributed data structures. We believe that the fos project has great promise as an alternative way to build systems which can scale up to meet the challenge of future multicore architectures. Some distributed data structures that the fos team believes are needed include a key-value store, a key value store which has a range based match function, a priority queue, and a data structure which can broadcast a global scalar.

I would like to explore more uses of dPool inside of fos system services. Another area that is worth exploring is whether dPool and the current dPool implementations are a good fit for fos running across multiple machines. fos currently provides the capability to extend messaging between machines in a single fos instance. It would be interesting to see if the additional latency involved with messages transiting between machines limits the scalability of dPool and whether algorithms that work well on a

single chip will extend well across multiple machines or if other algorithms will need to be developed.

Last, we are interested in mapping fos and dPool to future architectures which contain native message passing hardware. By mapping the fos messaging interface to this hardware, messaging cost can be reduced. It would also be interesting to see how this different messaging implementation would effect the scalability and performance of the dPool. One advantage of hardware messaging is that it would remove messaging load from the memory networks which could be helpful, especially for cases like we tested in this thesis where memory bandwidth was at a premium.

Bibliography

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

[2] 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.

[3] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 151–162, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[5] Jonathan Appavoo. *Clustered objects.* PhD thesis, Toronto, Ont., Canada, Canada, 2005. AAINR07602.

[6] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. Technical Report RC22863, International Business Machines, July 2003.

[7] Arvind and R.S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *Computers, IEEE Transactions on*, 39(3):300–318, mar 1990.

[8] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *Software Engineering, IEEE Transactions on*, 18(3):190–205, mar 1992.

[9] Henri Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A distributed implementation of the shared data-object model. In *In USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 1–19, 1989.

[10] J.-P. Banâtre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Gener. Comput. Syst.*, 4:133–144, September 1988.

[11] 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.

[12] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, pages 282–293, June 2000.

[13] 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.

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

[15] Adam Belay. Message passing in a Factored OS. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2011.

[16] Adam Belay, David Wentzlaff, and Anant Agarwal. Vote the OS off your core. Technical Report CSAIL Technical Report MIT-CSAIL-TR-2011-035, Massachusetts Institute of Technology, 2011.

[17] 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.

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[19] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[20] 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.

[21] 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.*

[22] Georges Brun-Cottan and Mesaac Makpangou. Adaptable replicated objects in distributed environments. Technical Report BROADCAST TR No.100, ESPRIT Basic Research Project BROADCAST, 1995.

[23] 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.

[24] Roy Campbell, Garry Johnston, and Vincent Russo. Choices (class hierarchical open interface for custom embedded systems). *SIGOPS Oper. Syst. Rev.*, 21:9–17, July 1987.

[25] 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.

[26] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[27] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31:314–333, 1988.

[28] Christian Clmenon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on multiprocessors. *IEEE Transactions on Software Engineering*, 22:152, 1993.

[29] Bram Cohen. Incentives build robustness in BitTorrent, 2003.

[30] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93. Proceedings*, pages 262–273, nov. 1993.

[31] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. Applebe, J. M. Bernabeu-Auban,

P.W. Hutto, M.Y.A. Khalidi, and C. J. Wileknloh. The design and implementation of the Clouds distributed operating system. *USENIX Computing Systems Journal*, 3(1):11–46, 1990.

[32] 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.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the ACM Symposium on Operating System Principles*, October 2003.

[34] 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.

[35] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[36] 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.

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

[38] Philip Homburg, Leendert Van Doorn, Maarten Van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An object model for flexible distributed systems. In *In Proc. First ASCI Annual Conf*, pages 69–78, 1995.

[39] 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, feb. 2010.

[40] Intel Corporation. *Intel Threading Building Blocks Reference Manual*, 1.14 edition, 2009.

[41] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[42] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, November 2000.

[43] Herbert Kuchen and Katia Gladitz. Parallel implementation of bags. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 299–307, New York, NY, USA, 1993. ACM.

[44] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

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

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

[47] Mesaac Makpangou, Yvon Gourhant, and Jean pierre Le Narzul. Fragmented objects for distributed abstractions. In *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, 1992.

[48] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.

[49] U.M. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-core 64-thread 64b power-efficient sparc soc. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 108–590, feb. 2007.

[50] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[51] J.W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, chapter POOMA: A framework for scientific simulations of paralllel architectures, pages 547–588. MIT Press, 1996.

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

[53] Adrian Schpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[54] Karsten Schwan and Win Bo. Topologies distributed objects on multicomputers. *ACM Trans. Comput. Syst.*, 8:111–157, May 1990.

[55] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 198–204, 1986.

[56] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Cline Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2:287–337, 1991.

[57] J.L. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, Changku Hwang, Hongping Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A.S. Leon, and A. Strong. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 98 –99, feb. 2010.

[58] Livio Soares and Michael Stumm. Flexsc: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1 8, Berkeley, CA, USA, 2010. USENIX Association.

[59] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 10–10, Berkeley, CA, USA, 1995. USENIX Association.

[60] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

[61] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 335–344, New York, NY, USA, 2011. ACM.

[62] Gabriel Tanase, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL pArray. In *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture*, MEDEA '07, pages 73–80, New York, NY, USA, 2007. ACM.

[63] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal

Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL parallel container framework. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 235–246, New York, NY, USA, 2011. ACM.

[64] Gabriel Tanase, Chidambareswaran Raman, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Languages and compilers for parallel computing. chapter Associative Parallel Containers in STAPL, pages 156–171. Springer-Verlag, Berlin, Heidelberg, 2008.

[65] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, 1993.

[66] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert Van Renesse, and Henri E. Bal. The Amoeba distributed operating system-a status report. *Computer Communications*, 14:324–335, July 1991.

[67] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 558–563, May 1986.

[68] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffman, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[69] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. The architectural design of globe: A wide-area distributed system. Technical Report Technical Report IR-442, Vrige Universiteit, 1997.

[70] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 98–99, 589, February 2007.

[71] VMWare, Inc. *VMCI Sockets Programming Guide for VMware Workstation 6.5 and VMware Server 2.0*, 2008. http://www.vmware.com/ products/beta/ws/VMCIsockets.pdf.

[72] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua,

Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86 93, September 1997.

[73] 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.

[74] 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.

[75] 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.

[76] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.