Scalable Applications on a Factored Operating System

Christopher R. Johnson
crjohns@csail.mit.edu
Charles Gruenwald III
cg3@csail.mit.edu
Nathan Beckmann
beckmann@csail.mit.edu
Harshad Kasture
harshad@csail.mit.edu
Anant Agarwal
agarwal@csail.mit.edu

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

1. INTRODUCTION

We will soon be seeing computer chips with hundreds, possibly thousands, of cores on a single piece of silicon. Traditional monolithic operating system kernels were designed for uniprocessor systems, and there are many issues with scaling them to large manycore and cloud systems [4]. We are exploring a naturally scalable operating system design which scales to large manycore systems and simplifies the creation of high performance, scalable applications in the cloud. The Factored Operating System (fos) is a scalable operating system built from the ground up to support cloud and manycore systems [4]. Operating system services are factored out of the kernel and into concurrent, distributed sets of user space processes called fleets. The design of fleets is naturally scalable to support cloud and manycore systems [5]. Cloud infrastructures are targeted by providing a single system image to user processes. Each process is presented a view of a single system, even though the system may be distributed across multiple physical machines in a datacenter and/or cloud. Figure 1 shows the fos system architecture. Each core executes applications and libraries on top of a fos microkernel. Processes communicate by message-passing using a distributed name service for locating services and capabilities for access control.

fos builds on traditional microkernel design ideas from Mach [7] and L4 [6]. It is differentiated by distributing and parallelizing high-level functionality within each OS service rather than simply exploiting parallelism between servers providing different services [4]. fos also is similar to prior distributed operating systems such as Amoeba [3] and brings the same concepts to multicore as well as emphasizing parallelism within each service.

2. SYSTEM DESIGN

fos was designed to target manycore and cloud systems by emphasizing scalability and a single system image abstraction. This section provides an overview of our previous work on these designs and how they relate to our application goals.

2.1 Fleets

Operating system services in fos are factored out of the kernel and into fleets. A fleet is a cooperating, spatially-distributed set of user processes that together provide an operating system service. Applications have POSIX system calls translated into messages to fos services, which run independently on distinct cores as fleet members. Previous work on this project describes how the design of fos supports the creation of fleets [5].

Fleets have three core design principles: First, fleets are scalable by design. Second, fleets are self-aware, monitoring and adapting their behavior through load-balancing and migration. Third, fleets are elastic, expanding and shrinking to match changing demand.

Each of these design principles benefit applications. Scalability helps to increase concurrency. Self-aware fleets can attempt to give each application a nearby provider of each
service it uses and move relevant data closer to the application. Elasticity ensures that there are enough fleet members to satisfy demand.

Our previous results demonstrated the benefits of fleets to user applications [5]. The throughput of the Linux page allocator, for example, was shown to collapse as the number of cores increased, while fos was shown to scale well using a fleet of page allocators (Figure 2).

2.2 Single System Image

Existing Infrastructure as a Service (IaaS) systems, such as Amazon Elastic Compute Cloud (EC2) [1] do not provide a uniform programming model for communication or allocation of resources in the cloud. They provide resources in the form of Virtual Machine (VM) instances and Linux kernel images [4]. Applications that require additional resources must be written to communicate with other machines manually.

fos builds on top of VM instances to provide a single system image to applications. Each process uses an application interface that resembles a single machine while the interface itself is implemented across several machines in the cloud or data center. The single system image allows applications to be written without knowledge of the actual infrastructure.

One challenge to providing a single system image implemented with fleets is the management of shared state. For example, distributed web servers in the cloud may share a single listening socket, and network fleet members must coordinate to ensure that no connection is accepted more than once. fos maintains a list of connections per network stack (i.e. machine) to achieve correctness and performance.

Figure 3 illustrates the single system image. App2’s processes are distributed across three physical machines in a compute cloud, and our POSIX implementation transparently forwards messages to provide the illusion of a single machine [5].

3. APPLICATIONS

Our current work is to port real applications to fos in order to evaluate the performance and scalability of the system.

Applications use our POSIX compatibility library which supports commonly used POSIX functions. Porting applications currently involves expanding the fos POSIX layer to support more functionality and performance optimization. Most applications can be ported with minimal or no modifications to the program itself. fos emphasizes process-based rather than thread-based parallelism, so some applications may require modifications to use messaging instead of shared memory. Alternatively, we have a working implementation of POSIX threads, so applications which use shared memory between threads may be ported as-is, but they are constrained to the same machine in our current implementation.

Currently, we have ports of a web server and memcached-1.4.8 [2] as well as the SPLASH2 and PARSEC benchmarks. We are focusing primarily on internet applications used for the web, such as HTTP servers and caching systems, since they are the dominant cloud workload and require high scalability.

We are currently benchmarking performance using our ported programs, and we are using those results to evaluate scheduling and messaging strategies.

4. ADDITIONAL AUTHORS

Additional authors: David Wentzlaff (Princeton University, email: wentzlaf@princeton.edu), Lawrence Stewart (Quanta Research Cambridge, Inc, email: lstewart2@csail.mit.edu), Adam Belay (Stanford University, email: abelay@cs.stanford.edu), Jeff Ward (MIT CSAIL, email: jward@csail.mit.edu), Lamia Youseff (Google, Inc, email: lyouseff@google.com)

5. REFERENCES