Automatic Scaling of Cloud Applications via Transparently Elasticizing Virtual Memory

Ehab N. Ababneh

University of Colorado at Boulder, ehab.ababneh@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds

Part of the OS and Networks Commons

Recommended Citation

https://scholar.colorado.edu/csci_gradetds/180

This Dissertation is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Automatic Scaling of Cloud Applications via Transparently Elasticizing Virtual Memory

by

Ehab N. Ababneh

B.Sc., Yarmouk University, 2003

M.Sc., State University of New York at Binghamton, 2007

A thesis submitted to the
Faculty of the Graduate School of the University of Colorado in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
2017
This thesis entitled:
Automatic Scaling of Cloud Applications via Transparently Elasticizing Virtual Memory
written by Ehab N. Ababneh
has been approved for the Department of Computer Science

Prof. Richard Han

Prof. Eric Keller

Prof. Sangtae Ha

Date ____________________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
This dissertation addresses the topic of how to achieve elasticity of an operating system so that networked resources in the form of remote memory and computation can be scaled up, down and out to meet the dynamic workloads of today’s cloud applications. This dissertation shows that it is feasible to modify the Linux operating system to achieve transparent elasticity by implementing four key primitives: *stretching* of a process’ virtual address space across the physical memory of networked nodes; fine-grained *jumping* of process execution across the set of networked nodes participating in the stretched address space; *pushing* of memory pages to different nodes to create islands of locality; *pulling* of remote memory pages to satisfy a local page fault. The dissertation further evaluates the overall ElasticOS prototype and shows the benefit of the system compared to networked swap, as well as performs an analysis of the performance of individual components of the system.
Dedication

I dedicate this dissertation to my family, wife and children.
Acknowledgements

I wish to thank my PhD advisor Prof. Richard Han, who helped guide me through this challenging and complex multi-year project. I also wish to thank Professors Eric Keller and Sangtae Ha, who worked closely with me on the ElasticOS project. I further would like to thank my committee members Prof. Chris Heckman and Prof. Shivakant Mishra. I’m grateful for the assistance from a fellow PhD student, Zaid Al-Ali. I’d also like to acknowledge Khaled Alanezi, a former PhD student and always a friend. Thank you!
Contents

Chapter

1 Introduction

1.1 The Promise Of The Cloud ........................................... 1
1.2 Elasticity In Software .............................................. 2
1.3 ElasticOS .............................................................. 2
  1.3.1 Thesis Statement ............................................... 4

2 State of the Art in Elastic Systems .......................... 5

2.1 Background ............................................................. 5
  2.1.1 What Is Elasticity? ............................................. 5
  2.1.2 Transparency .................................................... 6
  2.1.3 Remote Memory .............................................. 7
  2.1.4 Coherence Protocols ........................................ 7

2.2 Related Work ............................................................ 7
  2.2.1 Distributed Shared Memory .................................. 7
  2.2.2 MPI ................................................................. 14
  2.2.3 MapReduce and Hadoop ................................... 16
  2.2.4 Partitioned Global Address Space ......................... 16
  2.2.5 Remote Paging .................................................. 17
  2.2.6 Process And Virtual Machine Migration ................. 18
2.3 Summary And Discussion .................................................. 19
  2.3.1 Today’s Horizontal Scaling ........................................... 20
  2.3.2 Today’s Vertical Scaling .............................................. 22
  2.3.3 Summary ................................................................. 23

3 ElasticOS Design and Architecture ................................... 24
  3.1 Motivation ................................................................. 24
    3.1.1 Improving The Locality Of Reference .............................. 24
    3.1.2 Improving Scalability ............................................. 25
    3.1.3 Improving Page Placement ....................................... 25
    3.1.4 Zero-downtime Vertical Scaling .................................. 26
    3.1.5 Summary ............................................................... 27
  3.2 Architecture ............................................................. 27
    3.2.1 Vertical And Horizontal Scaling In ElasticOS .................. 28
    3.2.2 Transparency ........................................................ 29
    3.2.3 Remote Memory .................................................... 29
  3.3 Design And Implementation: ElasticOS’s Major Features ......... 30
    3.3.1 Resource Discovery ................................................ 30
    3.3.2 Identifying Opportunities For Elasticity ......................... 32
    3.3.3 Stretching ............................................................ 32
    3.3.4 Multi-node LRU "Pushing" ......................................... 33
    3.3.5 Pulling ............................................................... 35
    3.3.6 Jumping ............................................................... 35
    3.3.7 State Synchronization ............................................ 36
  3.4 Implementation Challenges And Pitfalls ............................ 36
    3.4.1 Process Checkpointing And Restarting ............................ 36
    3.4.2 Timely State Synchronization ..................................... 38
3.4.3 Heterogeneity ................................................................. 38

4 Elastic OS Evaluation ......................................................... 40

4.1 Experimental Method and Setup ....................................... 40

4.1.1 Setup .......................................................................... 40

4.2 Horizontal Scaling Performance ....................................... 41

4.3 Performance of Elastic OS’s Primitives ............................. 41

4.3.1 Latencies .................................................................... 41

4.3.2 Jumping ...................................................................... 42

4.3.3 Pushing And Pulling ..................................................... 43

4.3.4 Memory Pressure ......................................................... 43

5 Future Work ........................................................................ 46

5.1 Operating System Support for Objects ............................. 46

5.2 Mitigating The Hidden Cost of Jumping ........................... 47

5.3 Optimized Page Placement .............................................. 47

5.4 More Sophisticated Jumping Decision ............................. 48

5.5 Multithreading ................................................................. 48

5.6 Reliability ....................................................................... 48

5.7 Larger Test Suite ............................................................. 48

6 Conclusion .......................................................................... 50

Bibliography .......................................................................... 52

Appendix ................................................................................ 58

A Memory Management In Linux ........................................... 58

A.1 Virtual Memory Organization ........................................... 58
A.2 Translating Virtual Addresses To Physical

.................................................. 59
Tables

Table

3.1 Properties of ElasticOS and related approaches. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.1 Latencies of ElasticOS’s Four New Primitives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Figures

Figure

1.1 Vision: ElasticOS ......................................................... 3

2.1 Elastic Operations. ..................................................... 6

2.2 Distributed Shared Memory Abstraction ............................ 8

2.3 MemNet Architecture .................................................. 11

2.4 Message Passing Model of Parallel Computing ...................... 14

2.5 Partitioned Global Address Space (PGAS) ......................... 17

3.1 Example Natural Groupings Within Nodes In A Graph. ............. 26

3.2 ElasticOS Architecture Highlight ................................. 28

3.3 ElasticOS Components ............................................... 31

3.4 Page Fault History Circular Buffer ................................. 35

4.1 Execution Time Of Bread-first Algorithm Under ElasticOS vs Nswap. 42

4.2 Execution Time For The Benchmark ............................... 44

4.3 ................................................................. 44

4.4 ................................................................. 44

4.5 Number Of Free Memory Pages On Node 1 Running Nswap .......... 45

5.1 One Page Housing Two Highly-Connected Nodes in Two Different Clusters In The Graph. 47

A.1 Organization of a typical process’s address space in Linux .......... 58
A.2 Page Table Entry in x86-64bit................................. 59
Chapter 1

Introduction

1.1 The Promise Of The Cloud

In the past few years cloud computing has been gaining popularity due to its ability to help organizations scale up/down their compute infrastructure to meet the requirements of dynamically changing workload characteristics. Before that, enterprises built their own data centers and had to over-provision in order to keep up with varying compute power needs. By sharing a pool of resources, cloud computing created the illusion of unlimited resources.

To understand the importance of this emerging technology consider an enterprise that offers a web-based service housed in a classical data center. It is very crucial for the service to handle varying volume sizes, forcing the enterprise to over-provision when planning for capacity to meet business needs. This over-provisioning renders the data center resources under utilized during off-peak hours, days, or months.

In contrast, consider the same service running in a cloud computing environment. It can respond to fluctuations of service demand by either one of two ways. First, it can *scale horizontally* by spinning up additional compute nodes to handle larger volumes during peak demand hours, and then, *scale down* by taking those additional nodes out of service and shut them down during off-peak hours. Second, it can *scale up* the service by migrating to a larger compute node during those peak hours, and *scale down* again by migrating back to a smaller one during off-peak ones. Certainly, this flexibility is made possible by the cloud’s promise of shared resources permitting on-demand compute power availability, which helps customers maximize their return on investment.
1.2 Elasticity In Software

Cloud computing offers the ability to use more compute power on demand, but it is up to the cloud application to actually benefit from this offering. And what can limit its ability to do so is its architecture. In the web service application mentioned above, one approach to achieve dynamic scaling is that more web servers can be run in newly spawned nodes and then load balancers would redistribute the workload among all active servers. That may not be the case, however, for legacy applications, such as simulation software, or large graph analysis. For elasticizing such classes of applications, major recoding efforts and reconfigurations are necessary.

Distributed systems research produced several works that seek to address this problem. Previous approaches such as distributed shared memory (DSM) [56, 50, 72, 73, 23, 75, 62, 49, 35], MapReduce [29], message passing interface (MPI) [64], partitioned global address space (PGAS) [10], and remote paging based approaches [67, 32, 39] can be fitted with machine hot-plugging and hot-removal capabilities to allow applications to scale horizontally. Also, process and virtual machine migration proposals can be used for vertical scaling. Adopting these approaches, however, either degrades the application’s performance, or involve significant efforts required for redesigning the application to support elasticity, that may render them infeasible or undesirable. Clearly, software remains a key problem to achieving elasticity today.

1.3 ElasticOS

This thesis addresses the challenge described above. It proposes offering elasticity as a generic service supported by the software system. This service allows applications to scale up, scale out, and scale down to accommodate changing workload needs in an automatic and transparent manner to software developers. This new service is implemented as an integral part of the operating system and allows processes to stretch the limits of available resources beyond one machine, while supporting process mobility between nodes to minimize performance loss stemming from resource distribution.

We introduce ElasticOS, a new operating system built on top of Linux, which is a realization of the vision put forward above. Figure 1.1 provides a high level view of ElasticOS. If each element of the
grid represents a node in a datacenter, then we see that an instance of ElasticOS is capable of spanning a number of nodes in the data center, and that the number of spanned nodes can elastically scale up or down depending upon application demand. This new operating system supports four new primitives: (1) stretching the address space of processes across a cluster of compute nodes, allowing processes to jump (2) from one machine to another within the set of nodes participating in the stretched address space, to maximize locality, pushing (3) pages between nodes for optimal placement, and pulling (4) memory pages to serve remote page faults. We term our approach of stretching virtual memory address space across multiple physical nodes as *elastic virtual memory*.

The focus of our efforts towards an elastic operating systems for cloud applications will be on jointly elasticizing memory and computation. Outside of the scope of our work is elasticizing other elements of the OS, such as I/O devices and file systems.
1.3.1 Thesis Statement

We can summarize the contributions of this thesis as follows:

- Demonstrate the feasibility of automated transparent elasticity: we implemented ElasticOS on top of the Linux kernel to support the four primitives of stretching, jumping, pushing and pulling, which collectively allow applications to automatically scale vertically, horizontally, and down in a transparent manner.

- Demonstrate that the transfer of execution, which we call jumping, is able to reduce network traffic and latency to access memory to an extent that outweighs its cost. In comparison to a main alternative, network swap [67], where swap space is backed by remote memory over a network, with ElasticOS we saw a 67% reduction in runtime.

- Evaluate the performance of individual components of ElasticOS.

In the remainder of this dissertation, we first describe related work that may help achieve software elasticity in Chapter 2. In that chapter, we also identify what is still missing in the state of the art. Next, in Chapter 3, we first describe our approach to address those gaps. This is followed by a detailed description of the architecture and components that were introduced into the Linux kernel to achieve ElasticOS. Chapter 4 describes our evaluation of ElasticOS, including analyzing the latency performance versus networked swap, as well as a performance analysis of individual components of ElasticOS. We conclude with a discussion of future work in Chapter 5.
Chapter 2

State of the Art in Elastic Systems

2.1 Background

Supporting elasticity is no easy task. Here, in this chapter, we would like to list some of the challenges that need to be addressed before achieving this goal and some of the possible design choices. Then, we will list all previous approaches that can enable cloud applications to elastically scale up/out/down. And lastly, we will discuss how these approaches may fail to achieve true elasticity.

2.1.1 What Is Elasticity?

In cloud computing, elasticity is a crucial property which describes the system’s ability to quickly accommodate changing workload variations by provisioning and de-provisioning resources automatically, such that at each point in time the available resources fit the current demand as closely as possible. This property is often confused with scalability. The latter is another property that describes the system’s ability to accommodate larger workloads, which assures the performance of cloud applications. This could be achieved, as mentioned before, by either scaling out or up. Essentially, elasticity combines the methods of scalability with the ability to scale down or in when fewer resources are needed. See figure 2.1. Scaling down/in are equally important abilities; these are the economical aspect of elasticity which maximizes the return on investment for cloud customers while meeting their computational demand.
2.1.2 Transparency

Transparency is a design choice where the separation of components in a distributed system is concealed from the user and application programmer. So, the system is perceived as a whole rather than as a collection of independent components. This design enables the programmers to focus on the application logic rather than on the infrastructure it runs on, but it choice comes at a cost; little to no information is expected from them about the intended way of accessing data structures which could guide the distributed system infrastructure to optimally place the data for maximum exploitation of locality of reference.

There are different kinds of transparency that the distributed system may incorporate [26]. For example, users and software developers should be unaware of the distribution of resources. So, whether they are distributed or local, they should be accessed the same way, which is termed access transparency. Location transparency means that no extra code or steps should be taken when accessing remote objects, and users do not need to specify the location of a remote resource when accessing it. Migration transparency means that resources can be moved around in the distributed system and that should not impose any extra effort on software developers or users to deal with this resource migration. Scaling transparency means that a system should be able to grow without affecting application algorithms. Graceful growth and evolution is an important requirement for most enterprises. A system should also be capable of scaling down to small environments where required, and be space and/or time efficient as required.
2.1.3 Remote Memory

Stretching a process’s address space over a cluster of machine would result in page faults referencing memory pages residing on another machine. These remote page faults are very costly and should be minimized. One of the possible ways to reduce the effect of these remote page faults is to employ a smart page placement mechanisms, which can keep pages on the same machine where they are mostly used.

Another issue related to distributing a process’ address space is that page movements from one system to another need to be reconciled with the memory management units in the involved systems, since this is not a true symmetric multi-processing (SMP) system. This means that in some case moving pages around may trigger cache flushes and page table updates.

2.1.4 Coherence Protocols

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

2.2 Related Work

Given the context above on elasticity and topics to consider in achieving it, we now discuss process scaling approaches, including DSM (section 2.2.1), MPI (section 2.2.2), MapReduce and Hadoop (section 2.2.3), PGAS (section 2.2.4), and remote paging (section 2.2.5).

2.2.1 Distributed Shared Memory

DSM folds many physically-separate memory banks, installed on neighboring machines, under the umbrella of one virtual address space. See figure 2.2. This is made possible by the operating system’s
virtual memory manager (VMM) that decouples virtual from physical memory. Mapping virtual to physical addresses is described in appendix [A].

The general idea behind DSM is that when a process tries to acquire more memory the OS will try to satisfy the request locally first. And if that is not possible, the OS will have to look elsewhere by querying other machines in the cluster or consult status information cached locally. Once a satisfying chunk of memory is found, the two machine will exchange messages in order to commit this allocation, and process resumes.

On a page fault, the OS will consult the process’s page tables and locate the page table entry (PTE) that triggered the fault and whose value indicates whether the page is located remotely or locally. If the page was remote, a special module will handle fetching the page. Otherwise, the page fault handler resumes normally.

Over the years there have been many DSM proposals and implementations. They vary in their choices of coherence protocol, networking protocols, or others. Next, I will list some of their most important details.

2.2.1.1 Appllo\Domain

Apollo OS was an operating system used back in the late 1980s. Apollo\Domian was designed as a shared file system for a cluster of personal machines running Apollo OS. Its concepts can be easily extended to DSM [55].

In its core design, Apollo employs a distributed network-wide object sharing module called Object Storage System (OSS). OSS assumed a flat address space where each object has its own unique identification.
(UID). Analagous is the case with virtual memory address spaces where the page frame is the shared object.

Each object has a “home”. This can be the machine where it is first allocated. When an object is updated, it is given a time stamp that is carried with the object whenever it is copied. Updates to a particular page must applied to it at its home machine. OSS will check if the update carries with it the current object’s time stamp. If both time stamps agree, the update is committed, otherwise, it is rejected.

OSS also provides locking mechanisms where an object can be locked for read or write. All lock/unlock requests are sent to the home machine. Replies return with the object’s time stamp. Other copies of the object may be invalidated. Lock requests are enqueued. So, failed ones are retried if necessary.

2.2.1.2 Ivy

Ivy [56] [50] (short for Integrated shared Virtual memory at Yale) is a DSM built using Apollo OS workstations connected via a token-ring network. As in Apollo, Ivy assumes a flat virtual address space. Its smallest manageable unit, however, was a page.

In Ivy, each process has its own local address space, and another globally shared one. The local section of the memory is accessible only by the owner process, while the shared section is visible to all processes in the system. A shared-virtual-memory manager enforces the rules associated with sharing the common portion.

A page in Ivy can have one these properties: read only, read/write, or nil (i.e. invalid). These properties are used to enforce these rules:

- A page is read/write on one machine, nil on all others.
- A page is read-only on one or more machine, read/write on none.

These rules can be enforced by following these steps:

- Read fault on an invalid page:
  - Demote read/write (if any) to read-only.
  - Copy page
* Mark local copy read-only.

- Write fault on an read-only page:
  
  * Invalidate all other copies.
  
  * Mark local copy read/write.

For locating pages in the system, Ivy employs a central manager that keeps track of page ownership and locations. Page faults must query it, which then will decide whom should receive page-invalidation or page-forwarding requests. Machines that receive page-invalidation request, simply invalidate the page, and machine receiving page-forwarding request forwards the page to the one where the fault happened.

### 2.2.1.3 Clouds

Clouds [72] is a distributed operating system developed at Georgia Tech. It makes clear distinction between memory and threads. The first is organized as a collection of objects, which encapsulate data and code that will operate on it. Threads are not associated with a particular address space. They can, however, access these objects.

A collection of objects form an address space. Threads traverse address spaces of objects they access. Each object is comprised of one or more segments, and each segment may contain one or more pages. Pages are the unit of sharing.

A core module, called distributed shared memory controller (DSMC) on each node owns and manages the local segments. DSMC also provides mechanisms for read/write locking that can be used to access objects.

Threads can access objects in one of four modes: read-only, read/write, weak-read, and none. Read-only mode provide non-exclusive lock while read/write provides exclusive locks on the segments. None mode provides exclusive access to the segment, but does not lock it. Any new request would result in the segment being taken away to service the request. Lastly, weak-read provides read access to the segment where strong consistency is not required.
2.2.1.4 Mach And Agora

Mach [73] is a multiprocessor operating system developed at CMU. To a large extent, it provides the same abstractions found in a modern Linux kernel [57], namely tasks, threads, pipes (named ports in Mach) messaging, and virtual memory maps (memory objects). It is designed to run on tightly-coupled processors similar to Non Uniform Memory Architecture (NUMA) machines.

Agora [23] was built to extend Mach and provide shared memory semantics on a network of machines. It allows processes on different machines to share data structures over the network. The original copy of the data structure is the master, and the replicated ones are cached copies. All updates to the data structure are applied on the master and then replicated to the cached copies. This is similar to copy-on-write semantics.

2.2.1.5 MemNet

MemNet [30] was designed at the University of Delaware as a memory abstraction over a ring network. It allows sharing chunks as small as 32-bytes. It also devised a specialized hardware (MemNet device) whose purpose is to accelerate remote memory access. The insight behind this device is that a software stack executed to copy or update remote memory incurs significant performance overhead.

Each node in the system consists of a host and a MemNet device. See Figure 2.3. The host has access to its private memory, which is inaccessible from other nodes. MemNet devices also have on-board memory which is partitioned into two spaces; cache and reserved. Reserved chunks are guaranteed storage locations (i.e. can not be invalidated for memory reclaim purposes), while the cache chunks may be invalidated to free
up space during high memory contentions. There is also a tag table on each device to maintain the validity, "reservedness", and exclusivity of each chunk.

On a read fault, the valid bit is checked and if it is set the fault is satisfied locally. Otherwise, MemNet devices communicate among each other to locate and fetch the required chunk. Exclusive bit is set to 0. On a write fault, if exclusive bit is set to 0, a chunk-invalidate packet is sent all other devices, exclusive bit is set, and the fault is then satisfied.

2.2.1.6 Choices

Choices [75] is an operating system developed at the University of Illinois at Urbana-Champaign and written in C++. Its code adheres to object-oriented programming principles, which makes it easy to customize by replacing subframeworks and objects. For example, the virtual memory abstraction is represented in the Domain class.

Choice’s memory management is similar in principle to the one found in Mach. It organizes processes’ memories into objects (MemoryObject class). A process’ virtual memory, which is an instance of Domain class, maintains information about MemoryObject instances. Sharing memory between process is achieved by mapping the same MemoryObject by two or more different domains. They can also be shared across the network using an instance of DistributedMemoryObjectCache class. It is responsible for serving page fault requests for remote pages. It is also responsible for invalidating copies of a page when it is written to.

Choices also allows page locks to allow mutually exclusive access to a page by denying other accesses until the lock is released. While Choices manages memory as objects, its unit of sharing is a page.

2.2.1.7 Mether

Mether [62] is a set of mechanisms built on top of SunOS v4.0. It does not have a built-in mechanism for maintaining consistency. Instead, it provides a set of mechanisms exposed to the software developers by system calls. And it is up to them to decide and build the coherence protocol that meets their needs.

Mether provides three ways to update a page. First, the process that last updated a page can explicitly
update all other copies. Second, a process holding a page can invalidate its copy of a page if it receives a notification update from another process, this will trigger a page fault the next time the page is referenced. And third, a process can immediately ask for the page to be sent if it receives a page-update notification.

2.2.1.8 Munin

Munin [49] is another project that incorporates several techniques to improve DSM’s performance by reducing the amount of communication required to maintain consistency compared to previous DSM systems. To achieve this goal it supports multiple consistency protocols to allow different applications to choose the one that best suits their needs depending on the access patterns. It also supports multiple concurrent writers which addresses the problem of false sharing by reducing the amount of unnecessary communication performed keeping falsely shared data consistent.

2.2.1.9 TreadMarks

TreadMarks [52] is another DSM implementation done at Rice University. The authors first built a DSM system, and then compared its performance in environments with varying network speeds. They concluded that DSM is a viable technique for parallel computation on a cluster of workstations given a very fast networking infrastructure.

Then, they proposed two major features that build on their observation. The first is reducing the amount of communication performed to maintain memory consistency. So, they equipped TreadMarks with a lazy implementation of release consistency, which allows multiple concurrent writers to modify a page, reducing the impact of false sharing. And the second was using a standard low-level protocol, AAL3/4, bypassing the TCP/IP protocol stack.

2.2.1.10 FaRM

Fast Remote Memory (FaRM) [35] is the most recent implementation of a DSM system done at Microsoft. It leverages RDMA[^1], which supports reliable reads and writes of remote memory, and can do

[^1]: RDMA is short for Remote Direct Memory Access.
Figure 2.4: Message Passing Model of Parallel Computing

so faster than TCP/IP over Ethernet because it bypasses the kernel, avoiding performance overheads incurred by protocol stacks, all that without involving the remote CPU since this is directly supported by NIC cards.

FaRM provides lock-free reads to accelerate the common case usage scenarios and RDMA writes to implement a fast message passing primitive. It also wraps those reads, writes, and other memory operations such as allocate and free in transactions with ACID\textsuperscript{2} guarantees under strict serializability.

2.2.2 MPI

DSM systems

During the early 1990s researchers and industry representatives met to draft a standard for SPMD\textsuperscript{3} method of parallel computation. The result was MPI \cite{1}. The first draft of the MPI standard \cite{6} was published in 1994. Now, it is in version 3.1 which was published in June 2015.

In MPI, a single source code is duplicated and ran on a cluster of machines. Each process maintains its own private address space and communicate with processes in the system using point-to-point, or collective methods of communications. The software developer is responsible for partitioning the workload among all workers and for deciding when and how they are to communicate. This design allows MPI to make little to no assumptions about the underlying hardware. Thus, providing more portability and flexibility.

Listing 2.1 shows sample MPI program written in C. The code computes the summation of all el-

\textsuperscript{2} ACID is short for Atomicity, Consistency, Isolation, and Durability.
\textsuperscript{3} SPMD is short for Single Program, Multiple Data.
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    int world_rank, world_size, i, total = 0;
    int arr[] = {6,2,3,9,4,5};
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    printf("Current Rank: \%d\n", world_rank);
    if (world_rank != 0)
    {
        int tempTotal = 0;
        for (i = 0; i < 3; i++)
            tempTotal += arr[(world_rank - 1) * 3 + i];
        printf("Rank \%d sending value \%d\n", world_rank, tempTotal);
        MPI_Send(&tempTotal, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Rank \%d value sent\n", world_rank);
    }
    else
    {
        int tmp;
        printf("RANK 0 receiving values:\n");
        for (i = 1; i < world_size; i++)
            MPI_Recv(&temp, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Received value \%d from rank \%d\n", temp, i);
        total += temp;
        printf("Sum of the array = \%d\n", total);
    }
    MPI_Finalize();
}
```

Listing 2.1: Example MPI Program. Compute The Summation Of All Array Elements
ements in an array. Notice the function call MPI_Comm_rank at line 10 to find the rank of the current process. This is a sequence number given to each one of the participating processes and is typically used to partition the workload among all of them as is the case in this program; see the extra logic at line 15. Also, notice the calls to MPI_Send (line 18) and MPI_Recv (line 24). This is an example of point-to-point message exchange.

2.2.3 MapReduce and Hadoop

MapReduce [29] is a programming model and a framework that distributes workload across a cluster of machines. It performs two main operations, map and reduce, on large data sets. The map operation applies a function (e.g. sort of filter) on a set of data items and produces another more refined set of items. The reduce operation, then, applies another function (e.g. summation) on the output of first step to produce a summarized result. Underneath these two operations is a distributed framework that handles network communications such as for shuffling data, process creation/restarting/termination, and others. Hadoop is the open-source version of MapReduce.

2.2.4 Partitioned Global Address Space

Partitioned Global Address Space (PGAS) [28] is a parallel and distributed programming model that assumes a global address space logically partitioned among processes in the system. Each process has affinity to its allocated section of the global memory, but does not own it, thereby exploiting locality of reference while sharing it with other processes. In addition to the global address space, each process owns its private one. PGAS is supported by language extensions such as UPC [4] and a language-independent networking layer that provides communication primitives tailored for SPMD languages and libraries called GASNet [12].

Figure 2.5 depicts the general principle behind PGAS. Each object in the global domain is managed by one process. Accesses to remote objects are trapped and relayed to GASNet transparently. Process-object affinities are assigned by the software developer via annotations in the source code.

As in MPI, the software developer is concerned with partitioning the task into independent units,
2.2.5 Remote Paging

Researchers explored expanding available virtual memory by grouping idle memory resources in computer clusters to form a big swapping space for the computers that are short of memory resources. Here are the most prominent of such works.

2.2.5.1 Nswap and MemX

Network-backed swap was proposed to extend available virtual memory to machines when under pressure. Examples of this approach are Nswap [67] and MemX [32]. In these approaches a virtual block device is added to the running system and is then used as a swap device. The drivers then handle receiving block I/O requests and relaying them to a multi-threaded server running on a remote machine, which will cache the data on the local physical memory. These approaches assume that disks are the major contributer to the time spent in swapping pages in and out of the storage medium.

There is a slight difference between the two, however, in that MemX extends the Xen hypervisor while Nswap is designed as a kernel module injected into a running operating system.
2.2.5.2 Infiniswap

Infiniswap [39] is in many ways similar to MemX. It is also designed to be a virtual block device exposed by a hypervisor to the operating system running in a virtual machine. But it differs in that the client (i.e., block device) interacts with the cache server over RDMA. The insight here, is that RDMA over Infiniband is faster than NVMe, which is the fastest storage technology.

2.2.6 Process And Virtual Machine Migration

Process and virtual machine migration is employed in order to scale up the process or VM by moving it to another machine with larger physical resources.

2.2.6.1 MOSIX

Work on MOSIX started back in 1977 [18], and since then it went through many cycles of development to support newer hardware and newer operating systems. The first version was called Multicomputer OS (MOS), and it was based on Bell Lab’s Seventh Edition Unix and tested on PDP-11/45 and a diskless PDP-11/10 that were connected by parallel I/O. The name was changed to MOSIX in 1989, and the last version was released in 2006 and was called MOSIX-1 which ran on x86 workstations and servers connected by 1Gb Ethernet and was based on Linux 2.6.

It provided aspects of a single-system image. So that, users can login on any node and do not need to know where their programs run. It did not require modifications to the code or linking it with special libraries. It also supported process migration so that processes can migrate from slower to faster nodes or nodes with more memory [20].

2.2.6.2 Sprite

The Sprite Operating System [11, 66] was an research Unix-like distributed operating system developed at the University of California, Berkeley by John Ousterhout and his group between 1984 and 1992. As in MOSIX, it supported single system image on computer clusters. And it was the first to introduce log-structured file system.
This work was intended to provide transparent network awareness to the operating system, so that a process can migrate from one node to another in the cluster. It also offers the possibility of remote paging so that processes can benefit from the availability of physical RAM on remote nodes in the cluster.

2.2.6.3 Kerrighed

Kerrighed [6,16] is a single-system-image (SSI) operating system for clusters of workstations. Kerrighed offers the view of a single SMP machine on top of a cluster of standard compute nodes. Processes running in Kerrighed will have the ability to migrate from one node to another, checkpoint and restart, use distant memory, etc. Kerrighed is implemented as an extension to the Linux operating system.

In many ways, Kerrighed is similar to the above mentioned Sprite OS with the addition of limited DSM concepts.

2.2.6.4 CRIU

Checkpoint/Restore In Userspace, or CRIU [3], is a software tool for Linux. It enables freezing a running application (or part of it) and creates a checkpoint of it, which can be then used to restore the application and run it exactly as it was after the instant of freezing. This ability to checkpoint and restart the application can be used in live migration, remote debugging, and many other tasks.

CRIU checkpoints an application incrementally while allowing to run simultaneously. This is achieved by storing an initial copy of memory pages, and setting memory protection flags on previously copied segments in order to trap later modifications, which will then be applied to the initial copy.

2.3 Summary And Discussion

Despite the advancements in distributed systems, true elasticity remains elusive. Here we will discuss how previous approaches can achieve elasticity, including scaling down, not just scaling up and out, and what is still lacking in each of these approaches.
2.3.1 Today’s Horizontal Scaling

2.3.1.1 DSM

DSM can support horizontal scaling if it is fitted with machine hot-plugging and hot removal capabilities. When scaling out, a new node can be added to the existing set of nodes and then the elasticized process is allowed to use the newly available physical memory. And when scaling down, a victim node can be chosen and its share of memory pages is redistributed among other nodes. All that can be done transparently to the software developers.

While DSM can support transparent elasticity, it failed, however, to achieve acceptable performance levels to make it desirable. This is especially true for applications that exhibit high rates of memory writes since they trigger coherency messages to guarantee consistency.

2.3.1.2 Parallel Processing Frameworks

MPI can theoretically facilitate some sort of support for horizontal elasticity by dynamically spawning new worker processes or shutting them down. But practically, this is hard to achieve since it was first designed to use a fixed number of resources. MPI versions 1.x could not specify any means for changing the number of processes during the execution. While this changed with MPI version 2.0, this feature is not yet supported by many of the available MPI implementations. Moreover, due to its design, achieving this task requires significant effort to redistribute the data among the new process group. Thus, existing MPI programs are not designed to vary the number of processes. Adaptive MPI [44] addresses this issue by allowing flexible load balancing across different numbers of nodes, but requires complete application refactoring and significantly degrades performance when load balancing is not needed. MPI also is not well suited for cases where applications exhibit a lot of data intra-dependencies.

PGAS faces the same challenges above. It too supports a fixed number of processes. And since PGAS shares the same principles in relation to workload distribution, it is also expected to face the same problems if or when dynamic process support is implemented.

MapReduce is too-limited a framework for many workloads. It is designed for systems handling large
amounts of independent records typically found in log processing applications. Moreover, it is certainly not transparent to the software developer. For example, weather simulation applications require significant efforts to fit into the MapReduce model.

2.3.1.3 Network Swap-Based Approaches

Swap-based approaches may also be able to achieve transparent horizontal scaling since they extend the virtual memory available in a compute node by using remote physical memory for paging. Such solutions, may increase the effective node’s memory capacity automatically, but they fail to provide the adequate performance level needed for memory-intensive applications. For that they fail to provide a solution to the thrashing problem [31].

Imagine a compute node running a memory-intensive application. If the node’s available physical memory is fully utilized, further page allocations will force existing pages to the swap space, be it disk or remote memory. This overflow could result in subsequent page faults requesting swapped out pages. In such cases, the process will have to be scheduled out of the CPU while waiting for access to the page to be restored. This could also involve evicting another victim page. Whether the page is swapped out to network or to a hard disk drive, in neither one of those cases the process can avoid being scheduled out of the CPU, forcing it to lose precious CPU time.

It is important to understand that fetching a page from swap space involves more than simply the latency of the storage medium or network. It also involves this loss of CPU time described above in addition to time needed to execute the software stack that handles fetching the page including device driver code, which is also non-negligible. In fact, one of the reasons that NVMe [9] standard was proposed is to reduce the time spent in accessing storage in relative to the SCSI standard.

The above mentioned problems are inherent to the remote-paging-based approaches. They partition local and remote physical memories into two classes; with local memory being the first-class, and the remote memory being a second-class resource. With this separation, there is always a need to keep as much of the memory in the first class as possible, and only use remote resource when absolutely needed. If applications are truly to benefit from remote memory, there should be away to use it as if it was local.
In contrast, ElasticOS addresses thrashing by permitting execution to jump in a fine-grained manner to a remote machine, thereby turning the remote memory into local memory. We explain this in more detail in the following chapter.

2.3.2 Today’s Vertical Scaling

Clearly, process and virtual machine migration can support vertical scaling by migrating the cloud application to a compute node with more resources.

To mitigate the cost of migration, live process and VM migrations typically employ an iterative copying technique, which stretches the cost over a longer period of time. The procedure is outlined here for process migration:

- **Step 1**: Copying Meta data. In this step process descriptors are copied and used to create the replica shell.

- **Step 2**: Copying process data. Here, the content of physical pages are copied and injected into the replica shell. Pages modified after being copied are recopied.

- **Step 3**: Atomic Handover. Finally, the process is frozen. Then, the very last pages are copied, and control is transferred to the new process replica.

Proper migration in these approaches depends on the correctness of the atomic handover in the third step. Without it, proper functionality cannot be guaranteed. The time needed to accomplish this step is certainly going to be a downtime imposed on the application. And it is length depends on the effectiveness of the process copying performed in the second step. It also depends on the application.

Freezing the process in these techniques is necessary for correct functionality, since the process’s identity is not allowed to stretch over multiple machines. However, if the process was allowed to have a distributed identity, then this downtime could, theoretically, be reduced to zero.

Also, as we mentioned before vertical scaling is always limited by the available capacity of the host machine.
2.3.3 Summary

In summary, as it stands today, existing techniques like DSM, parallel processing frameworks, network swap, and migration-based vertical scaling, are unable to achieve fully automatic and transparent scaling up/out/down of a process’s memory and computation.
Chapter 3

ElasticOS Design and Architecture

For applications with uneven usage, or spikes during different time periods, having built in elasticity and scalability is crucial. However, most legacy applications have been developed to run on a single machine and require recoding to adapt for both the scalability and elasticity that the cloud provides.

ElasticOS is a modern operating system for the cloud, which addresses this challenge by supporting elasticity as a generic service. Its design aims to allow processes to automatically scale and achieve seamless vertical and horizontal scaling.

3.1 Motivation

ElasticOS introduces four new primitives, namely stretching, jumping, pushing, and pulling. Here, we will discuss how these address the limitations in prior work described in Chapter 2.3.

3.1.1 Improving The Locality Of Reference

Imagine an application that is iterating several times over a working set size of data larger than what can fit in physical memory. Normally, it will thrash (i.e., move pages in and out of memory) constantly. ElasticOS, deals with this scenario by allowing the process to jump (i.e., transfer execution) to the remote machine where some of the required data is stored in memory when it detects that "too many" pages are being pulled from the remote machine. Caching reveals that there is temporal locality in the referencing of data (and code). This locality is reflected spatially in the storage of pages on remote nodes, creating "islands of locality" of pages stored on networked nodes. Jumping of execution allows ElasticOS to jump to an island
of locality on another node, and achieve considerable execution while operating primarily on the pages in
that island with limited network overhead of pushing and pulling pages outside that island. This is a very
profitable trade-off since jumping involves copying very few memory pages.

Another scenario that can show the power of jumping is when two processes on two different nodes in
a DSM system contend for write access on a memory page. This scenario will force the coherency protocol
to constantly invalidate and copy the page between the two nodes. This is in contrast to ElasticOS, which
will allow one of the processes to jump closer to where the other process is so both of them can contend
locally for the page via local synchronization. In this way, ElasticOS avoids the non-scalable invalidate-copy
operations on that page characteristic of DSM-like approaches.

### 3.1.2 Improving Scalability

Fine-grained jumping of execution can potentially provide better scalability compared to DSM and
remote-paging/network-swap based approaches. Jumping treats all shared memories from the all nodes as
a first-class resource. For example, consider a process stretched over two machines. Forcing it to jump
from one to the other after pulling too many pages will turn remote memory into local memory, thereby
effectively increasing the pool of available memory seen as local physical memory by the process. Network
swap-based techniques only move memory back and forth, not execution, and hence cannot exploit islands
of locality during execution to reduce network overhead.

### 3.1.3 Improving Page Placement

Jumping also opens the door for optimizing page placement within the system. Remote-paging-
based approaches and DSM leave no options but to keep as much memory as possible in the local machine.
ElasticOS can employ a multi-node page placement algorithm, where natural groupings of memory pages
resulting from pattern of reference can be exploited, namely islands of locality.

For example, when traversing the graph shown in figure 5.1 a good page placement may group nodes with
the same color on the same machine so that when traversing an edge that connects brown nodes to one of
the yellow nodes, execution can transfer to the machine where the latter are located.
Improving page placement tries to maximize locality of reference, which is discussed above, but on a larger time scale.

### 3.1.4 Zero-downtime Vertical Scaling

Stretching in concert with jumping, pulling and pushing can enable zero-downtime vertical elasticity by amortizing the cost of atomic handover discussed in section 2.3.2 since it allows the process to distribute its identity over multiple machines. That is, ElasticOS typically first stretches the address space of a process to span the memory resources of another machine, and eventually fills that stretched node’s memory with pages through normal pushing and pulling. At some point, execution of the process jumps to the stretched remote node when it is beneficial to do so, namely when there is an island of locality to exploit on the remote node. This approach naturally lends itself to vertical scaling when the remote node is substantially more resource-rich (or poor) than the initial machine. ElasticOS’s approach can be seen to be more general and flexible than conventional process and VM migration approaches discussed in section 2.2.6, which confine the migrated process or VM to one presumably larger target machine, limited to the resources of the target machine. In contrast, ElasticOS permits a process or VM to span not only the target machine but other machines nearby. Also, unlike standard migration approaches, there is no need to freeze the process for a large time frame during an atomic handover phase from one machine to another.
3.1.5 Summary

Having described the major properties of ElasticOS, we present the following table that summarizes the benefits of ElasticOS compared to other approaches to elasticity described earlier in related work.

Table 3.1: This table compares the properties of ElasticOS to related work.

<table>
<thead>
<tr>
<th></th>
<th>ElasticOS</th>
<th>DSM</th>
<th>MPI</th>
<th>PGAS</th>
<th>MapReduce</th>
<th>Remote Paging/Network Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertical Scaling</strong></td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Horizontal Scaling</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Transparent</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Coherency Protocol</strong></td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Explicit Messaging</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

*ElasticOS uses jumping to maintain coherency, which is more scalable than messaging.

3.2 Architecture

In this section, we will outline the architecture of ElasticOS at an implementation-independent level, describing its goals and highlighting its functionalities. Then, in section 3.3, we will describe OS-specific details.

ElasticOS provides a single virtual address space, which can map physical memories from different nodes. Every node is identical, containing several modules as shown in Figure 3.2, each of which has a different function. In addition, ElasticOS enables execution to jump among machines that are contributing their physical memories to the single virtual address space.

The system monitor continuously collects processes’ memory usage parameters to detect variations in the real-time demand. And when it finds such a process whose active memory set size is too big to fit into RAM, it issues a command for starting a new VM instance and instructs the multi-node scheduler to stretch this application over the two nodes.

Stretching, which resembles a remote fork, is initiated by a special signal sent to the check-point module, which creates a snap shot of the process, bundles it, and sends it to the restart module in the remote node. This module, then, uses the snap shot data to create a clone of the original process. The application,
then, resumes and its clone remains in a suspended state.

With the stretching done, the process’s presence is extended to multiple nodes. At this point, a multi-node LRU page balancer starts to select pages from the home node and pushes them to the slave. The pages server receives page push requests and injects them into the duplicate process’s address space. Additionally, any modifications to the process’s meta data, such as new memory mappings or files opened, are also applied to the clone, this is performed by the synchronization module.

To maximize the opportunities for exploiting locality of reference, ElasticOS tries to keep the elasticized process near its "better" part of memory. So, the system monitor continually observes the process’s page faults. If one of those references a remote page, the fault is trapped and forwarded to the elastic page fault handler, which pulls the page from the pages server and restores access to it locally; also, it updates its internal per-process page fault history to record the occurrence of this fault type. Over time, the collected history information will provide a picture of where the process’s memory accesses are going. At that point, the multi-node scheduler can make an informed decision and choose to either keep the process running locally, or instruct the checkpoint module to force the process to jump to the remote machine. That should result in more machine-local memory accesses and, thus, better performance.

3.2.1 Vertical And Horizontal Scaling In ElasticOS

When vertical scaling is desired ElasticOS can resize the compute node by starting a new instance that is larger than the one currently in use and stretching the application to it. And with more aggressive
page balancing, migrating a large bulk of the process’s memory may be accelerated. Then, once the process jumps to the larger node, it can be pinned there. This will force all remaining pages to be pulled from the original node, thus, migrating the whole process to the larger node. And once all these steps are done, the original node can be safely shutdown. In contrast to other proposals, ElasticOS’s approach for vertical scaling will not impose any significant down time since it operates while the process is active.

Vertical scaling is limited by the fact that you can only expand to the physical size of the server. Beyond that, horizontal scaling is a must. Additionally, cloud service pricing policies may favor a set of small machines over one large machine in terms of cost efficiency. In such cases and others, horizontal scaling may be more feasible and favorable.

ElasticOS’s framework can support horizontal scaling by stretching the target process address space to the newly started node. And as before, then page balancing and process jumping kick in to maximize locality and balance the load on both machines.

3.2.2 Transparency

Unlike other proposed distributed operating systems like fos [80] and Barrelfish [21], ElasticOS can be built on top of a commodity operating system. And the next section describes how it can be built on top of the well-known Linux kernel, which in essence eliminates the need to adopt new programming models. Furthermore, ElasticOS does not require any modifications to the source code, which is in contrast to many of the other approaches including MPI, PGAS, MapReduce and others. To compensate for the lack of information passed from the software developers via programming language annotations or specialized API, instead, ElasticOS monitors system events and memory reference patterns to infer key information that can help partition the memory footprint of the elasticized process in an optimal way.

3.2.3 Remote Memory

Most operating systems use multi-level page tables [15] to map virtual addresses to their physical counterparts. Those are organized in a tree-shaped index, with the page table entry (PTE) as leaf nodes. Each PTE maps the starting address of a virtual page to a physical page frame number if a valid flag is set. If
this flag is reset, this means the page is not valid (i.e., has not been allocated or swapped out), in which case the operating system handles fetching this page from the swap or create a new mapping to a fresh physical page. ElasticOS, which is built on top of Linux, extends this by creating a virtual swap device that maps physical memory of another node as a swap space. So, each node in the system of N nodes will have at least N-1 virtual swap devices. This approach allows an elasticized process to quickly find a remote page and fault it in whenever it is referenced by reducing the steps needed to find a remote page.

When a page is sent over the wire upon a request from another node, the page is reclaimed, and an invalid PTE entry is placed into its place in the process’ address space. This entry is also a handle to the page’s new location. These steps guarantee that only one copy of each page is active at any given time. This design choice has the effect of eliminating the need for an expensive coherence protocol.

### 3.3 Design And Implementation: ElasticOS’s Major Features

This section discusses key features of ElasticOS and how the OS-independent architecture described in the previous section is implemented in our prototype within the Linux kernel.

#### 3.3.1 Resource Discovery

Whenever a machine starts, it broadcasts a message on a pre-configured port announcing its readiness to share its resources with elasticized processes. The message will include two groups of information. The first is connectivity parameters, such as IP addresses and ports. The second group will list the machine’s available resources, which include total and free RAM. The same information will also be used for load balancing purposes.

Once a node receives such a message from another node, the two nodes establish socket connections that will be later used for direct communication needed for page balancing and process check point data exchange.
Figure 3.3: ElasticOS Components.
3.3.2 Identifying Opportunities For Elasticity

Elasticizing a process can be achieved either manually or automatically. The manual way is achieved by invoking a command line tool and supplying it with the process id of the target application. The automated way involves monitoring the overall system’s RAM utilization, and then examining the memory usage of all processes in the system to trigger elasticization.

Linux maintains memory utilization indicators called watermarks. There are three levels of them: min, low, and high. These levels drive the kernel swap daemon’s (kswapd) activity. When memory usage reaches the high watermark, page reclaim starts, and when it goes down to low watermark, page reclaim stops. ElasticOS leverages these watermarks and the level of kswapd’s activity to detect periods of memory pressure.

To identify opportunities for elasticity, the system monitor searches for memory-intensive processes, and when it finds one it marks it for elasticity. These processes have the property of maintaining a large working set of physical memory page frames and high rate of page swap-ins. This property can be identified by examining the per-process counters Linux maintains to keep track of memory usage. These variables include: 1) task_size inside the type struct mm_struct which keeps track of the size of mapped virtual memory, 2) total_vm inside the same structure to count physical memory pages allocated, 3) rss_stat of type struct mm_rss_stat which contains an array of counters including one for the number of swapped out pages, and 4) maj_flt variable inside the struct task_struct which counts the number of swap-ins triggered by the process.

3.3.3 Stretching

Once a process is identified as a target for elasticity, it is stretched to another node. As we mentioned before, stretching is a form of remote fork where process meta data is sent over the network to the remote restart module instead of being copied from one memory location to another. The restart module then handles allocating the various data structures needed to create the duplicate process and fill them with the data it receives from remote check point.
A process (the parent) is forked in Linux to create a child process by duplicating its descriptor (i.e., an instance of `struct task_struct`), and then a few fields in the newly created child process’s descriptor are assigned new values to give it its own identity. Such unique per-process fields and data include CPU context (i.e., instance of `struct thread_struct`), auditing information, stack region, and a few others. Perhaps the most important field is the memory descriptor (an instance of `struct mm_struct`), which contains information about the process’ virtual memory. Once this process "copying" step is done, the child is kicked into life by assigning it to a CPU.

Stretching resembles forking broadly described above, but it needs to compensate for the "remoteness" of the process’s home. So, a pre-configured baby sitter process is selected to be the new parent of the duplicate process on the remote machine. Also, the page table, which maps virtual page addresses to physical page numbers on the home machine, is not copied; instead, a new page table is created. That carries with it the assumption that all pages’ contents need to be faulted in from the home machine whenever referenced on the remote slave. Lastly, it is important to keep the child process in suspended state for awhile. This is to make sure that only one of the process copies is running at any given time, and in this case it is better to keep the process in the home machine running to maximize locality.

### 3.3.4 Multi-node LRU "Pushing"

A key challenge for ElasticOS is to balance memory pages of elasticized processes between machines in a way that minimizes the frequency of remote page faults, thereby reducing network overhead and latency. For this purpose we extend Linux’s page replacement algorithm, which is a second-chance LRU, by adding multi-node page distribution awareness to it. In this version, pages selected for swapping out belong to elasticized processes and are pushed to another node and injected into the address space of the process’ duplicate there. This is in contrast to the plain version of the algorithm where pages would be written into the swap device.

In second-chance LRU, a system’s pages are managed in a FIFO linked list. When a page fault occurs, the newly referenced page is placed at the beginning of the list; this automatically moves the existing pages back by one position. Since only a finite number of positions are available in the FIFO queue, the system
must reach its capacity limit at some point or another. When it does, the pages at the end of the queue are considered for swap out, but before that they are given a second chance. Each page is assigned a special field containing a bit that is controlled by the hardware. When the page is accessed, the bit is automatically set to 1. The software (kernel) is responsible for un-setting the bit. When a page reaches the end of the list, the kernel does not immediately swap it out but first checks whether the aforementioned bit is set. If it is, it is unset and the page is moved to the start of the FIFO queue; in other words, it is treated like a new page that has been added to the system. If the bit is not set, the page is swapped out.

From this broad description of how second-chance LRU works, you can clearly see that it tends to group pages in reference-based chronological order within the pages list. So, it is most likely that pages at the rear of the queue, which are typically considered for eviction, are related in terms of locality of reference.

One challenge that needed to be solved to implement page balancing is identifying pages belonging to an elasticized process and what virtual address they are mapped to. Luckily, Linux maintains a functionality called reverse mapping, which links anonymous pages to their respective virtual area map. By walking this chain of pointers and then finding which process owns that map, we can tell them apart from other pages owned by other processes in the system. Then, with simple calculations we can find the starting virtual address of that page.

As for moving pages from one machine to another, we created a virtual block device (VBD) that sends page contents using a socket connected to a page server on the other machine (VBD Server) rather than storing it to a storage medium. This is shown in Figure 3.3. This virtual block device is added to the system as a swap device. All pages belonging to an elasticized process sent to the other machine are allocated swap entries from this device. This swap entry is inserted into the page table of the elasticized process where the page is mapped. As a result, if that page needs to be faulted in later on, the swap entry will route the page fault to our VBD. This design choice allows us to reuse Linux’s page eviction and faulting code.
3.3.5 Pulling

As described above, when a page belonging to an elasticized process is offloaded to another machine its matching PTE is replaced with a swap entry that points to the virtual swap device. This guarantees that when the elasticized process faults on a remote page, Linux will be able to automatically route the request to the correct machine and serve it.

Page fault history for elasticized processes is recorded in per-process circular buffers, as shown in Figure 3.4. On a local page (denoted L in the figure), a marker is inserted and the local page fault counter is incremented. If this insertion takes the place of a previous page fault, as is the case when the buffer is full then the counter corresponding to the deleted entry is decremented. Similar steps are taken on a remote page fault (denote R). This tracking of page fault history will guide the multi-node scheduler as we will see next.

3.3.6 Jumping

As page fault history builds up, it will show the tendency of where page faults are "going". If, within a given window of page faults, remote faults start to accumulate, then the multi-node scheduler could determine that the process would better exploit locality of reference if it jumps to the remote node. So, it will reschedule the process to the other node.

Jumping starts by sending a special signal to the target process, which is handled by an in-kernel checkpoint module. This module will, then, copy only the necessary information for the process to resume on the other node. These information include: 1) the thread context, which includes the register state, 2)
pending signals, 3) auditing counters, and 4) top of the stack. Other information about the process will synchronized using a special module described next. These information are sent to the restart module in the other node, which will write each one of them in its proper location.

Once the checkpoint and restart modules finish exchanging the data, the checkpoint module suspends the process on its side, and the the restart module resumes the one on its side. This will guarantee that only one clone of the process is running at any given instance.

3.3.7 State Synchronization

Whenever the kernel modifies in-kernel data structures that belong to an elasticized process, a special message is broadcast to other nodes carrying all the necessary information to enable them apply the same change on the clones. Such events include mapping new memory regions, un-mapping ones no longer needed, opening new files, modifying existing file descriptors, and others. All such synchronization messages must be flushed to the network before next jump happens.

Extra care must be taken when jumping from one machine to another, though, as all state synchronization messages must be flushed to the network from the buffer before jumping takes place. This is necessary to maintain a consistent view of the resources.

3.4 Implementation Challenges And Pitfalls

Here we discuss some additional challenges we had to overcome when implementing ElasticOS features in the Linux kernel.

3.4.1 Process Checkpointing And Restarting

Efficient process checkpointing, which is needed for stretching and jumping, is necessary for ElasticOS since it is considered as a source of overhead. For these purposes we implemented it as a Linux kernel function invoked from a kernel thread to avoid context switches. This design choice posed a challenge since almost all kernel functions, with only a few exceptions, were designed to operate on the "current" process.
This made it necessary to refactor kernel functions that access process descriptors, map/unmap memory regions and many others.

Second, for obtaining an accurate checkpoint of the target process we needed to access the register state. This proved a very challenging task since processors do continuously expose them to the operating system’s kernel. In fact, and as noted here [79], on 64-bit x86 architecture, they are exposed in these three states: (1) during the return from an exception, (2) performing a system call, and (3) when handling interrupts. Therefore, the process must be in one of these three states when performing a checkpoint. Another requirement is that the process must be woken up to perform the checkpoint even if it was sleeping or suspended. Third, the process must be in a state where it can not run in user-space code to avoid race conditions.

Luckily, signal handling state on our chosen architecture satisfies all of the above mentioned requirements. So, we implemented process checkpointing as an in-kernel signal handler.

3.4.1.1 What is in the Checkpoint

Appendix A highlights Linux’s memory management. Figure A.1 shows how Linux organizes the virtual address space of a typical process.

- Code from binary file is loaded into the text segment. This need not be included in a check point under the assumption that nodes in ElasticOS share the file system where the binary file is saved. Simply when a process is stretched, the new shell copy can map the file with the same path.

- Data segment typically contains constants (e.g., strings) found in the binary file. For the same reason as above this segment need not be included in the check point.

- BSS segment houses global and static variables defined in the source code. Processes typically modify this section at a low rate. So, it need not be included in the check point, but should be faulted in when needed.

- Heap segment is where dynamic memory allocations smaller than a configured value (128KB by
default in most Linux distributions) are mapped to. Memory allocation requests larger than that are anonymously mapped; see below.

- Memory mapping segment houses two different types of memory regions: anonymously mapped heap, and file mapped. Anonymously mappings are heap regions larger than 128KB, and shared library files.

- Stack segment is where the call stack region is mapped into.

Needless to say, check pointing a process must be fast. That is especially true for jumping, since it is performed more frequently. This means we need to copy only what constitutes a minimally valid checkpoint.

For stretching, we need to construct a valid process shell. Page contents will have to be faulted in on demand. This means shipping the process’ organization meta data will be sufficient. Specifically, we can reconstruct the process using descriptions of the various virtual memory regions that make up the process such as their boundaries, flags, and file names if any.

For jumping, we need to end up with a runnable process after reconstructing it. This means we need to include stack pages, which are modified at a very high rate, and register state.

### 3.4.2 Timely State Synchronization

The operating system’s scheduling and socket transmission buffering can interfere with state synchronization. For example, an elasticized process may map a new memory region, whose information will be broadcast to other nodes in the system. This broadcast message, however, may not reach all destination nodes in a timely fashion before the process jumps to another machine. This could result in unwanted results. For this purpose, we added code to guarantee timely transmission of UDP socket transmissions.

### 3.4.3 Heterogeneity

When a Linux process is loaded into memory, a collection of shared objects are mapped into its address space. These are libraries that your applications gets "for free" (e.g., vDSO [22]). They handle system tasks such as performing a system call and the like. Pointers to their functions and data are typically
copied to other libraries in the application. If a process migrates from one system to another where one of those libraries differs in version, this may result in undefined system state since functions pointers would not match. For this reason, we assume that the two nodes have matching file systems.
Chapter 4

ElasticOS Evaluation

This chapter demonstrates the following points about the performance and behavior of ElasticOS:

- For some algorithms common for memory-intensive applications, ElasticOS offers better horizontal scaling than that offered by remote memory paging with comparable available resources.
- Determines the cost of each one of ElasticOS’s four new primitives.
- Demonstrates the importance of jumping to ElasticOS’s performance.
- Demonstrates that ElasticOS mitigates the effect of memory pressure on the compute node.

4.1 Experimental Method and Setup

To evaluate ElasticOS, we will compare it to Kerrighed as a candidate for implementing vertical scaling, and to Nswap for horizontal scaling. We will also examine the contribution of each one of its primitives to the overall performance of ElasticOS. We do not think that we need to compare it against MPI, PGAS, or MapReduce due to the disparity of the core principles on which these approaches stand. Also, to the best of our knowledge, there is no working DSM implementation that can run on today’s hardware.

4.1.1 Setup

We will use Oracle VirtualBox as our hypervisor. We will run two such virtual machines in our test setup, with the master being termed the VM upon which the original process is started, and the slave being
termed the second VM into which the process is stretched. To run the operating systems and applications we used Oracle’s VirtualBox \[13\]. Nswap’s implementation is based on the network block device project (version 2.6.19) available for Linux \[8\], with a modified server that holds data in memory rather than a file on disk. As for Kerrighed, we ported its code to Linux kernel version 2.6.38.8, which is the same as used for our implementation of ElasticOS. The host was an Intel Xeon E5-2690 V4 machine.

Our main benchmark is an implementation of breadth-first search algorithm. We measured the time it needs to complete not including the time for loading the data from files. The same benchmark is used to compare ElasticOS against Nswap and Kerrighed. The size of data is roughly the same size as the physical memory available on one of the VMs, which is 4GB.

4.2 Horizontal Scaling Performance

Figure 4.1 shows that ElasticOS is three times faster than Nswap. Intuitively, we attribute this to the flexibility afforded by ElasticOS to change the machine upon which it is executing, in comparison to Nswap whose execution is pinned on one machine, though it is free to move pages among the networked machines. Allowing execution to roam enables ElasticOS to move computation towards islands of data locality, thus reducing network overhead compared to Nswap, which can only transport data towards the computation. Evaluating ElasticOS with a larger test suite remains as a future work. Even though unmodified applications can run on ElasticOS, they, however, need to be written in a special way due to the current lack of distributed file system support among other features.

4.3 Performance of ElasticOS’s Primitives

4.3.1 Latencies

We need to estimate how much time on average each one of those primitives needs to complete. Their individual performances are important to the overall performance of ElasticOS. Table 4.1 shows the average latency for each one of those primitives.

For stretching, collecting the checkpoint data and sending it over the network, it takes 7.39 millise
Figure 4.1: Execution Time Of Bread-first Algorithm Under ElasticOS vs Nswap.

seconds. At the received side, it takes slightly more than half a second to fork a new process, fill it up with checkpoint data, create memory mappings, and schedule it to the processor. This is needed once per run. This startup latency is largely masked from the execution of the process as the stretching occurs behind the scenes.

For jumping, creating the mini-checkpoint data and sending it to the network socket buffer takes 0.18 milliseconds, and then about another 4.9 milliseconds at the receive end to modify the execution context and stack region, and then wake up the process. In our sample test run, the process jumped three times.

For jumping and pushing, latencies were not surprising at all since these are dominated by the network. However, for their frequencies, we noticed that pulling was the dominant side in page balancing. Pushing moved a bit more than 11000 pages, equivalent to 44 MBs. Pulling moved more than 0.7 million pages, and that is equivalent to 2.7 GBs.

4.3.2 Jumping

To understand the importance of jumping to ElasticOS, we measured how its absence impacts the overall performance. So, we ran the benchmark under ElasticOS once with jumping fully enabled, and another with the process pinned on the master machine (i.e., no jumping allowed), and lastly with the process pinned on the slave after the first jump. The results are shown in Figure 4.2.

The results indicate that, indeed, jumping contributes positively. ElasticOS performs better when
Table 4.1: This table shows the latencies of stretching, jumping, pulling, and pushing. Please note that some are measured in seconds and others in milliseconds. It also shows (last row) how many times on average they were performed.

<table>
<thead>
<tr>
<th></th>
<th>Stretching</th>
<th>Jumping</th>
<th>Pulling</th>
<th>Pushing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Send Latency</strong></td>
<td>7.39 ms</td>
<td>180 ms</td>
<td>0.36 ms</td>
<td>0.43 ms</td>
</tr>
<tr>
<td><strong>Receive Latency</strong></td>
<td>520.43 ms</td>
<td>492 ms</td>
<td>0.0023 ms</td>
<td>0.0027 ms</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>1 (times)</td>
<td>3 (times)</td>
<td>994586 (times)</td>
<td>11233 (times)</td>
</tr>
</tbody>
</table>

jumping enables applications to migrate to their required data.

### 4.3.3 Pushing And Pulling

To better understand how both pulling and pushing contribute to ElasticOS’s page balancing we logged the number of free pages in both nodes once every second. This is shown in Figure 4.3 for the master node, and in Figure 4.4 for the slave.

Pushing contributes to the load balancing during the first few seconds of the benchmark’s execution when the master machine is overloaded. Collected logs and time stamps show that kswapd started to push pages approximately 50 seconds from the start of execution and slowed down after approximately 240 seconds. Once enough pages have been pushed to the other node, remote page faults start accumulate, forcing the process to jump. Also shown in the logs, is that the process before the 300th second. Then, it started to pull pages from the master machine to reach a steady state. This shows that pulling is the main contributor to balancing pages in the system.

### 4.3.4 Memory Pressure

Here, we would like to understand ElasticOS’s ability to mitigate the impact of memory pressure.

In the Linux kernel, the parameters that determine the level of activity of the kernel swap daemon (i.e., kswapd) are three watermarks. These are called pages_low, pages_min and pages_high. User-space applications invoke kswapd when the number of free pages reaches pages_min. This path is called direct reclaim. And when the number of free pages reaches pages_low, the kernel invokes kswapd in an attempt to
maintain pages\textunderscore high of free pages.

We repeated the experiment described in section 4.3.3 and plotted the result in Figure 4.5. Noticeably, ElasticOS is better at maintaining a "healthy" node, while not sacrificing performance. As you can see, for almost the whole duration of the benchmark’s execution, the number of free pages were down at a very low level.
Figure 4.5: Number Of Free Memory Pages On Node 1 Running Nswap.
Chapter 5

Future Work

The research presented in this dissertation answered some questions about achieving elasticity in cloud computing. However, there are many questions that remain unanswered. Here, we briefly describe some of the possible directions future research should explore.

5.1 Operating System Support for Objects

Object-oriented programming (OOP) [27] has evolved as the standard paradigm for building software in the last few decades, and most likely will remain so for the coming decades as well. The foundations of today’s widely used operations systems were establish back in 1990s. This gap is bridged by compilers that OO language constructs into streams of instructions understood by the operating system and hardware. This is needed in order for the final binary code to be compatible with what the operating system supports. Take for instance Executable and Linkable Format (ELF) [17] which is supported by Linux. It does not specify any method for supporting OO constructs awareness.

Embedding objects support in the OS can be crucially important for many reasons. Imagine a memory page that stores a very small and frequently accessed object while the remaining locations in the page are relatively "cold". In this scenario the page utilization is very low. With proper objects support, "hot" objects can be grouped in smaller number of page frames which could improve the overall system’s performance.

In the context of ElasticOS, imagine two highly connected nodes from two different clusters in a large graph as shown in 5.1 The presence of those two nodes on the same page will force it to be pulled from one machine to another too often. This is obviously a similar problem as found false cache-line
Figure 5.1: One Page Housing Two Highly-Connected Nodes in Two Different Clusters In The Graph.

sharing [48]. However, the solution presented for that problem is not applicable in our scenario, since they generally involved padding the structure that causes the cache lines to be falsely shared with unusable memory locations so that each one of those would fix exactly a cache line.

It must be noted that we are not arguing for a complete re-write of the OS kernel, but rather we arguing for some sort of object awareness, which can be implemented in existing kernels.

5.2 Mitigating The Hidden Cost of Jumping

Page balancing in ElasticOS can improve page placement in the system to minimize page pulls, but it is not expected to be perfect. Also, the application’s memory access pattern can further complicate the task. Indeed, when logging all page pulls and pushes under ElasticOS, we noticed few pages in certain applications constantly being dragged from one machine to another after each jump. This is the hidden cost for jumping, which could be mitigated with smart page pre-fetching. A small hash table that stores the pull count for each page pulled after a jump may help. If these counts grows neck-and-neck with the number of jumps, then the respective page is added as a candidate for pre-fetching.

5.3 Optimized Page Placement

We have see from the evaluation second that pulling is the main contributor to balancing pages across the nodes. This is means that the elasticized application will have to wait for most of the pages that are
moved between nodes. We would like to explore more aggressive page balancing schemes that do not rely heavily on operating system’s kernel swap daemon.

5.4 More Sophisticated Jumping Decision

Currently, ElasticOS relies on comparing local to remote page faults within the captured faults in a history buffer. We would like to explore other designs that can capture the tendency of where page faults are going. One possible design is implementing multiple circular buffers with varying sizes. And if, for example in a setup of three buffers, remote page faults outnumber local ones in the largest buffer but they do not in the smaller buffers, then that could indicate faulting in remote pages was just a transient phase. Thus, jumping is not necessary.

5.5 Multithreading

The initial prototype of ElasticOS presented in this dissertation focuses primarily on testing of single-threaded processes. A more advanced version of ElasticOS will be needed to test multi-threaded processes and VMs. Modification of shared state by multiple threads jumping to the same machine to modify common synchronized data will incur jumping overhead, and this will have to be compared to the coherency overhead that would occur in DSM-like paradigms.

5.6 Reliability

As ElasticOS stretches to more and more nodes, the stretched address space becomes more vulnerable to failures of the memory or computation at any one node. More research will be needed to make ElasticOS more robust to failure as it scales out, and may include techniques such as replication.

5.7 Larger Test Suite

We plan on testing a much larger suite of applications on ElasticOS in order to more thoroughly determine under what conditions and what types of algorithms are most suitable for elasticization as we
have implemented it in ElasticOS.
Chapter 6

Conclusion

This thesis started with the premise that elasticity is a crucially missed property in today’s cloud computing infrastructure. It allows cloud applications to scale up/down/in/out in order to meet varying workload sizes. Based on this premise we identified several considerations that arise in the design of a generic service that can support automatic elasticity, and evaluated existing work based on them. We concluded that elasticity is still to this day a far fetched goal. This conclusion led us to several contributions.

First, we introduced ElasticOS, a modern operating system for the cloud. It allows applications to stretch the boundaries of available resources beyond a single machine, and proved its feasibility by describing how automatic elasticity can be achieved via its four new primitives; stretching, jumping, pulling, and pushing. We also described how these primitives can be implemented in the Linux kernel.

Second, we proved the desirability of ElasticOS by: (1) describing how its four new primitives can reduce the overhead imposed by the resource distribution and scale better than existing work, (2) evaluating it against Nswap, and (3) evaluating the contribution of jumping to the overall performance of ElasticOS. We did find that the benchmark’s execution time is reduced by almost 67% when ran under ElasticOS compared to Nswap. We also found that the compute nodes do not suffer from memory pressure under ElasticOS, but do under Nswap.

Finally, we highlighted possible ways ElasticOS can be improved and some of the questions that are yet to be answered. We described how in-kernel object support can improve the performance of ElasticOS, help native programming languages such as C support objects, and maximize the memory utilization. We also described how we can further improve the performance of ElasticOS by implementing pre-fetching and
smarter method for deciding when to jump. And finally, we highlighted how it can support a wider class of application, which will allow us to evaluate ElasticOS more thoroughly.
Bibliography


[57] Linux. [Online; accessed 01-July-2014].

[58] Alan M. Mainwaring and David E. Culler. Active message applications programming interface and communication subsystem organization. Technical Report UCB/CSD-96-918, EECS Department, University of California, Berkeley, Oct 1996.


A.1 Virtual Memory Organization

Process descriptors in Linux are instance of `task_struct`. Inside that there is an `mm` field, which is a pointer to process’s memory descriptor, which is an instance of `mm_struct`. This structure maintains a list of virtual memory areas (VMAs which are instances of `vm_area_struct`.

Each VMA is a contiguous and homogeneous (i.e. same permissions and file mappings) range of virtual addresses; these areas never overlap. Each VMA also specify start and end addresses, flags to determine access rights and behaviors, and the `vm_file` field to specify which file is being mapped by the area, if any. A VMA that does not map a file is anonymous. Each memory segment above (e.g., heap, stack) corresponds to a single VMA, with the exception of the memory mapping segment. This is not a requirement, though it is usual in x86 machines. VMAs do not care which segment they are in.

Figure A.1: Organization of a typical process’s address space in Linux.
A.2 Translating Virtual Addresses To Physical

The processor consults page tables to translate a virtual address into a physical memory address. Each process has its own set of page tables; whenever a process switch occurs, page tables for user space are switched as well. Linux stores a pointer to a process’ page tables in the `pgd` field of the memory descriptor. To each virtual page there corresponds one page table entry (PTE) in the page tables, which in regular x86-64 paging is a simple 8-byte record shown in figure A.2

Linux has functions to read and set each flag in a PTE. Bit P tells the processor whether the virtual page is present in physical memory. If it is set, then the field page frame number is valid. If it is clear (equal to 0), accessing the page triggers a page fault, and the page number is not valid. Keep in mind that in the second scenario, the kernel can do whatever it pleases with the remaining fields. And in Linux, they can be used to store the value of the swap entry, which tells the kernel how to find the page’s content on the swap device if they were indeed swapped out before. The R/W flag stands for read/write; if clear, the page is read-only. Flag U/S stands for user/supervisor; if clear, then the page can only be accessed by the kernel. These flags are used to implement the read-only memory and protected kernel space, which could be used to trap writes to that particular page.