Elastic Process:
A Framework for Joint Disaggregation of Memory and Computation in Linux

by

Zaid Alali

B.Sc., Jordan University of Science and Technology, 2014

M.Sc., University of Colorado Boulder, 2020

A thesis submitted to the

University of Colorado in partial fulfillment

of the requirement for the degree of

Doctor of Philosophy

Department of Computer Science

2023

Committee Members:

Richard Han

Eric Keller

Eric Rozner

Sangtae Ha

Shiv Mishra

Alali, Zaid(Ph.D., Computer Science)

Elastic Process: A Framework for Joint Disaggregation of Memory and Computation in Linux

Thesis directed by Prof. Richard Han

Scaling is essential in cloud computing to accommodate the variable need for resources by different applications. Scaling is always associated with the challenge of distributing resources, and this challenge usually stems from the fact that the underlying operating system is designed to be monolithic. Past efforts have attempted to break the monolithic design of the operating system like "ElasticOS" and "LegoOS" and introduce distributed resource management primitives to the operating system. However, previous efforts either suffered from performance degradation or faced a severe development challenge that requires modifying the complex monolithic source code of Linux kernel. We propose Elastic Process; an auto scaling framework with a new approach to achieve joint disaggregation of memory and compute primarily in userspace. By using tools like "Ptrace" and "CRIU" we were able to build a prototype that demonstrates joint disaggregation of memory and computation. Our test results on macro applications and off-the-shelf application shows execution time performance improvement as well as network traffic reduction when compared to remote swap approach. Our results for macro application also shows more performance improvement when using multiple threads.

Keywords: Scaling. Ptrace. CRIU.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has been an integral part of the delivery of services through the internet. Scaling in the cloud is essential to support the variable load of different applications and services. In this thesis, we investigate the challenges and limitations of scaling in the cloud, and we propose a robust framework for joint disaggregation of memory and computation to support autoscaling.

Scaling in the cloud typically follows one of the following approaches: i)Vertical scaling ii)Horizontal scaling or iii)Diagonal scaling. Vertical scaling is achieved by adding more resources to the application, like migrating a virtual machine(VM) to a node with more memory and compute power[7, 45]. Horizontal scaling is done by adding more of the same software or hardware instances, for example: adding more threads for a web service to handle more clients[31, 7]. Diagonal scaling is a hybrid approach of both vertical and horizontal scaling[14].

As shown in Fig. 1.1(A), vertical scaling usually requires migration of the application to a node with more resources, resulting in an unavoidable downtime[18]. Numerous research projects have been working on minimizing downtime of VM live migration[5, 43, 2], where some approaches tend to follow a "post-copy" strategy[17], by migrating the minimum state required to run the VM, and later copy the rest of the data asynchronously. Other approaches do "pre-copy"[27], where the VM is copied while running, and when migration is triggered, only the delta of changes are copied. Recently, vertical scaling has been applied on units of execution that are smaller than VMs, for example: container migration is now widely used in the cloud[41, 37, 9]. Other vertical scaling approaches

are done on a process level; for example CRIU[12] is a well known tool used to migrate processes between nodes using checkpoint restore approach.

Since vertical scaling requires migration of the application to nodes with sufficient resources, migration usually goes in one direction; for example, an application running on node A scales vertically by migrating to node B with more resources, and once the migration is done, none of the application's state is retained on node A. In other words, vertical migration does not usually assume migration back to the same node.

In horizontal scaling adding more instances of the same software and hardware as shown in Fig. 1.1(B). Horizontal scaling is commonly used for load balancing[30], where a set of tasks are distributed over a set of resources like web servers load balancers[6]. Recently, serverless computing[4, 29, 10] platforms have been emerging in the cloud, these platform offer Functions as a Service (FaaS)[15, 20], where the unit of execution can be as small as a function. FaaS platforms highly utilize load balancing and horizontal scaling to distribute functions efficiently on different nodes. However, only a small set of applications are suitable for deployment on FaaS platforms due to limitation of service offering; limitations on execution time where functions have to be short lived and stateless[4].

Another common use of horizontal scaling is distributing batch jobs on multiple nodes, for example: Hadoop[3], Apache Spark[46] and MPI[48] are frameworks used to distribute threads of execution along with the corresponding data on multiple nodes for processing. However, distributing computation and memory places a heavy burden on developers since it requires application redesign and code refactoring to scale horizontally, in addition to having a good knowledge of the network and the underlying infrastructure.

Hybrid scaling approaches also known as Diagonal scaling[14, 23, 38] have been emerging recently, where both vertical and horizontal scaling is used. As shown in Fig. 1.1(C), typically diagonal scaling is done in two steps, first vertical scaling is done until the resource limits of nodes are reached, then horizontal scaling is applied. However, switching from vertical scaling to horizontal scaling requires the application to be horizontally scalable, like web and transactional applications, otherwise the application code must be refactored to scale horizontally.

All different scaling approaches suffer from many limitations; for example, horizontal scaling requires the application to be designed to horizontally scaled, or code refactoring

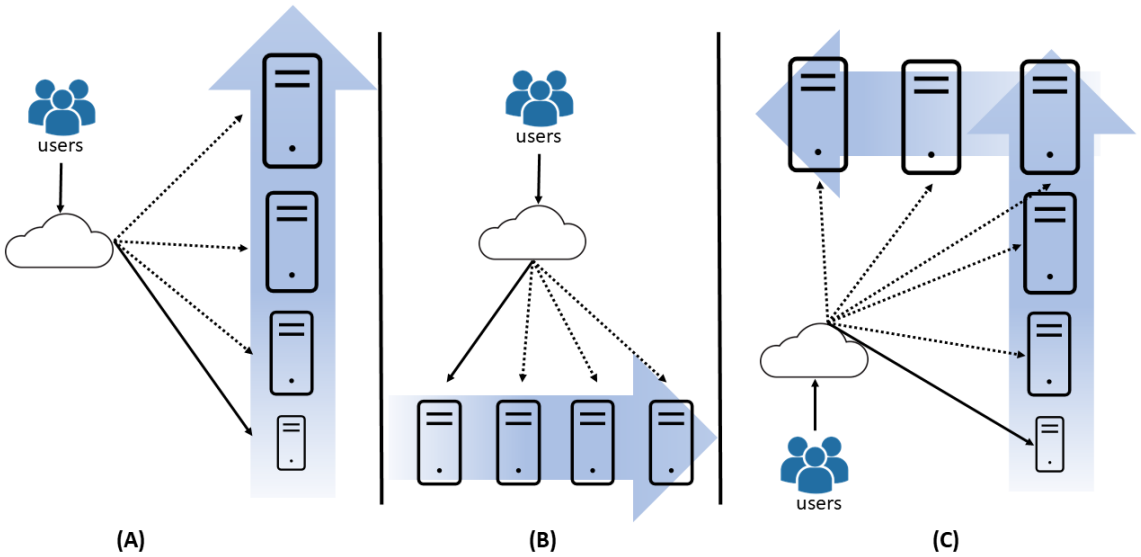


Figure 1.1: Scaling Approaches: (A)Vertical, (B)Horizontal, (C)Diagonal

must be done. Vertical scaling migrates the application to a node with more resources without retaining application state behind, thus, missing the opportunity to utilize any data locality that may exist at the node it migrated from.

The motivation of this thesis is to overcome some of the current scaling approaches challenges by enabling joint disaggregation of memory and computation in Linux. As shown in Fig.1.2 we present a framework that enables us to manage resources of multiple nodes, therefore, enabling a process to auto scale these resources transparently as if it was running on a single node.

Thesis Statement: This thesis demonstrates the feasibility and desirability of the Elastic Process concept; a primarily user space implementation of automatic scaling of a process and its threads joint disaggregation of memory and computation.

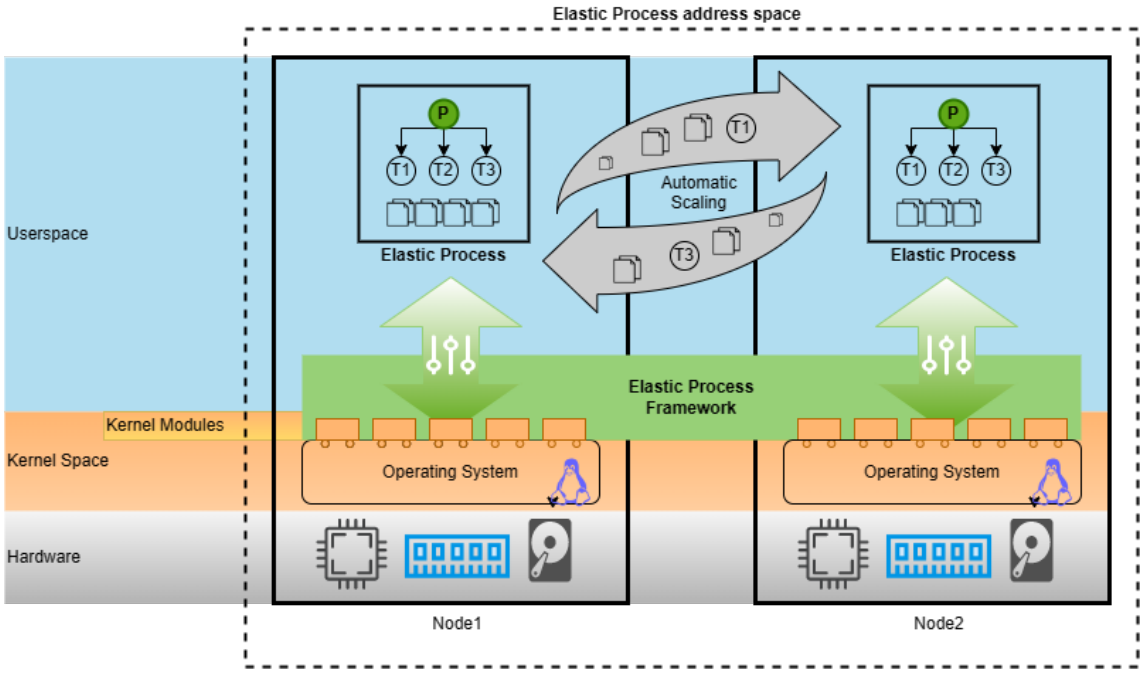


Figure 1.2: Elastic Process framework overview

This thesis makes the following contributions:

- Implements Elastic Process: a joint disaggregation of memory and computation framework for Linux
- Prove the desirability and stability of Elastic Process framework
- Provides performance evaluation of Elastic Processes across a variety of macro test applications and an off the shelf application.
- Evaluate disaggregation policies, including parameter sensitivity analysis
- Evaluate multi threaded implementation of Elastic Process versus single threaded.

Chapter 2

Related Work

In this chapter we will discuss related work on resources disaggregation; we will focus on memory and compute resources and previous work done in this area.

2.1 Memory Disaggregation

Memory disaggregation in the cloud has been widely explored[35, 26, 47]. Memory disaggregation approaches typically pin the application for execution on one node, and the memory footprint is distributed on other nodes, and the application has a remote direct memory access(RDMA)[11]. As shown below in Fig. 2.1, when Process A exhausts the memory on Node1, it uses network swap to free some space by swapping out to Node2. When the process needs to access a page from remote memory, more network swapping is required to access the needed page.

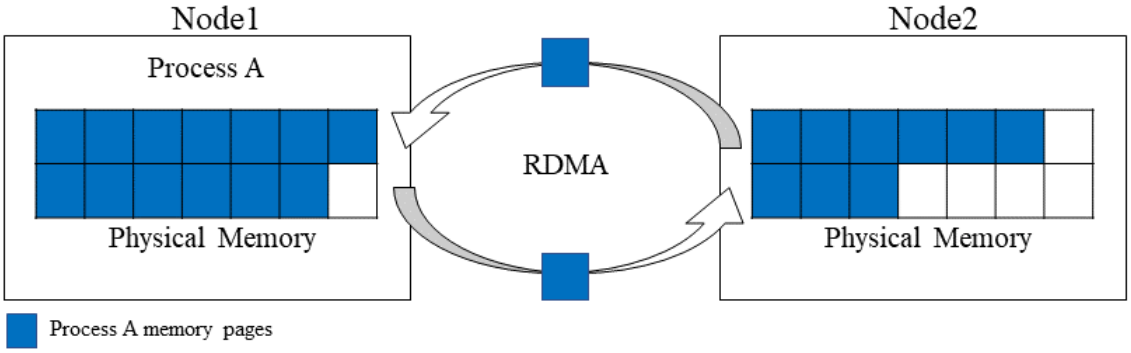


Figure 2.1: Network Swapping

Nswap[34] is one of the early research projects that enabled memory disaggregation

by building a network swapping module for Linux systems. Nswap enables a network block device that sends swapped out memory pages over the network via Ethernet to be stored on remote memory. Nswap evaluation results proved that swapping pages over the network with RDMA is 1.7x faster than using disk for swapping.

RDMA research areas have been exploring different network technologies; for example, Infiniband[25] has been utilized to improve RDMA for network swapping purposes. Infiniswap[16] showed that swapping over Infiniband can be up to 6x faster than disk swap and improves memory utilization of a cluster.

Memory disaggregation via RDMA improved swapping by minimizing swap latency and increased the efficiency and utilization of memory in data centers. However, network swapping approaches pin the compute on one node, in other words, data must follow compute. Therefore, missing a great opportunity of utilizing data locality on remote memory, as well as avoiding a lot of network traffic resulting of excessive swapping activity.

RAMCloud[36] is a low-latency storage system that aggregates the memory of multiple servers to appear as a single large coherent key-value store. RAMCloud achieves low-latency by storing data in DRAM with log structure mechanism for backup copies of the data in secondary storage. RAMCloud provides a seamless memory disaggregation capabilities to applications by using polling-based communication approach that bypasses the kernel and communicate directly with the network interface cards to access data on remote DRAM.

FluidMem[8] is a seamless memory disaggregation approach for virtual machines(VMs) in the cloud. Memory disaggregation in FluidMem is achieved by modular integration of remote memory backend like RAMCloud, which allows dynamic scaling of VMs to multiple machines as well as downsizing the VM's memory footprint to zero. FluidMem's modular integration of remote memory approach enables cloud operators to easily manage remote memory without cloud users involvement.

2.2 Computation Disaggregation

Computation disaggregation usually takes advantage of horizontal scaling frameworks and cloud offerings. Amazon Lambda[21], Microsoft Azure Functions[22], Google

Cloud Functions[28] and IBM Cloud Functions[39] are FaaS[15] offerings that enables computation disaggregation on a function level. FaaS as a computation disaggregation approach has many limitations and challenges; where functions must be short-lived, stateless and limited in size.

Other computation disaggregation approaches utilize data processing frameworks like Hadoop, Spark and MPI. As mentioned in Chapter 1, such frameworks enable developers to distribute threads on multiple nodes, allowing parallel processing and the results can be aggregated eventually on one node. In order to distribute computation on multiple nodes, developers must redesign and build applications that were never designed to run in a distributed manner. Also developers must figure out how to distribute threads of execution and data on different nodes, hence, data dependency can be a challenging problem since threads of execution and the corresponding data are pinned on a specific node until they are done executing.

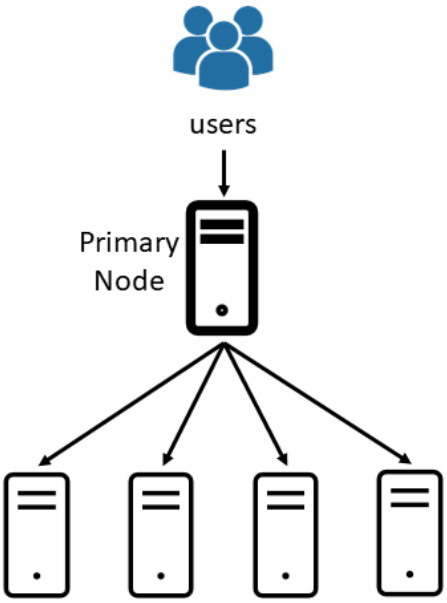


Figure 2.2: Typical Computation Disaggregation Architecture

Fig. 2.2 shows a typical computation disaggregation architecture. Disaggregated compute architectures commonly have a "Primary Node" responsible for distributing computations on different nodes, and collecting results at the end if needed. The primary node can have different names based on the platform or framework used for computation distribution; for example, on FaaS platform the primary node is called a "Dispatcher", and for web applications the primary node is called a "Load Balancer". Also, data processing frameworks like Hadoop and Spark call the primary node "Cluster Management" or "Master" node.

2.3 Joint Disaggregation of Memory and Computation

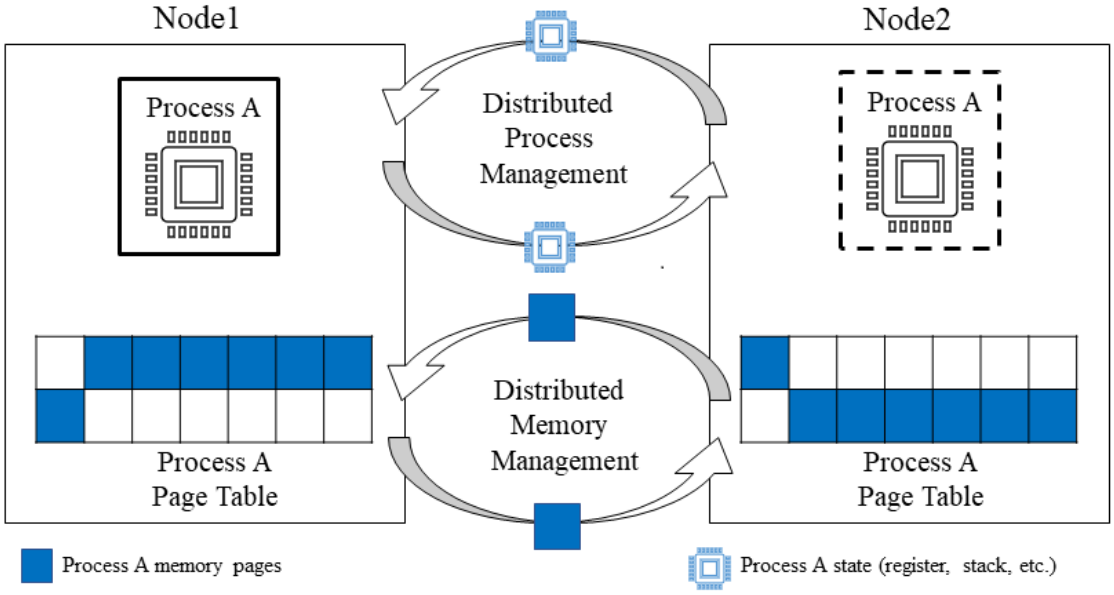


Figure 2.3: Joint Disaggregation of Memory and Computation

Distributing memory and computation discussed in Sec. 2.1 and Sec. 2.2 suffers from a crucial limitation; pinning either memory, computation or both on one node. When application compute or memory is pinned on one node, it could miss a great opportunity of utilizing resources available on other nodes in the cluster; for example, if computation was pinned on one node and memory was distributed, moving computation towards memory can substantially improve performance, especially if memory was distributed as islands of locality.

Fig. 2.3 shows a distributed process and memory management units, where Process

A utilizes both memory and compute resources on Nodes 1 and 2. Fig. 2.3 shows Process A state can be copied between nodes to enable execution and context switch on both nodes, as well as allowing access to pages of memory both locally and remotely.

The limitation of pinned compute and memory stems from the classic monolithic design of operating systems, where process and memory management units assumes that managed resources only exist locally on the same node. There have been many attempts to disseminate monolithic kernels in order to build a distributed operating system that enables management of resources both locally and on remote nodes.

2.3.1 LegoOS

LegoOS[40] proposed to break the monolithic design of the operating system by disseminating the operating system functionalities into dissaggregated network attached monitors. As shown below in Fig.2.4 LegoOS introduced a split kernel model that consists of hardware resource monitors, and appears to the user as a distributed set of server, where monitors can manage resources on multiple nodes by sending messages across hardware components over the network.

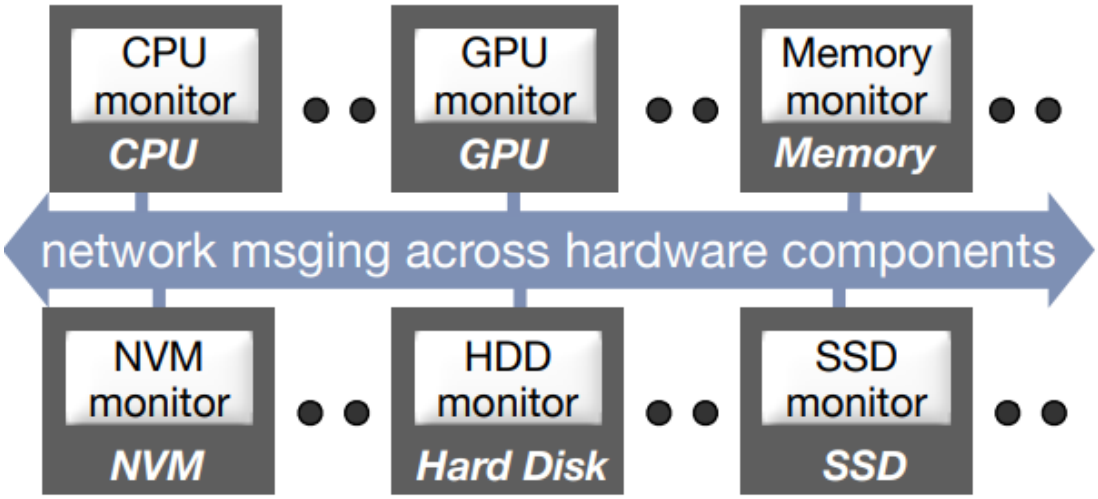


Figure 2.4: LegoOS Split Kernel [40]

LegoOS targets processor, memory and storage components and identifies them as `pComponent, mComponent and sComponent` respectively. For each applications LegoOS will use a `pComponent, mComponent and sComponent` where different components can communicate via infiniBand network messaging. Finally, the resources of different ma-

chines are exposed to the user as vNode, which is similar to a virtual machine.

While improving resource packing and lowering failure rates, LegoOS suffered from performance degradation when it comes to execution time of test applications. LegoOS splits hardware components on nodes into monitors that have to communicate and synchronize the state across the cluster, resulting in a very high overhead of 2X to 4X slower run time when compared to monolithic servers.

Testing and working with LegoOS have can be difficult and have the following limitations:

- LegoOS runs on emulated hardware, which makes testing more difficult and slow.
- There are hard requirements for a specific hardware set to run LegoOS[24].
- LegoOS pins compute and memory on selected `pComponent and mComponent`, and does not have agile execution transfer capability.
- Development, maintenance and debugging is challenging since LegoOS runs in kernel space.

2.3.2 ElasticOS

ElasticOS[1] enables joint disaggregation of memory and compute by modifying the Linux kernel 2.6 source code. Fig.2.5 shows the architecture of ElasticOS which implements a new set of operating system primitives; **Stretch, Jump, Push and Pull**. The **Stretch** primitive is used to create a clone of the target process on a second node, which allows the process to expand its address space on a remote nodes. ElasticOS reuses the kernel function `copy_process()` that lives in `fork.c` file in the kernel, where the entire process is cloned excluding the heap memory, which will be copied on demand using the Pull mechanism. **Jump** is a lightweight migration that allows a process to move and resume execution on a remote node without copying the entire address space. ElasticOS manually copies the process's `task_struct` data structure which contains the state information regarding the target process including CPU registers. The last step of Jump in ElasticOS is to update the return address of the last system call to resume execution from the last checkpoint, and this is achieved by modifying x86 assembly function `ret_from_fork`.

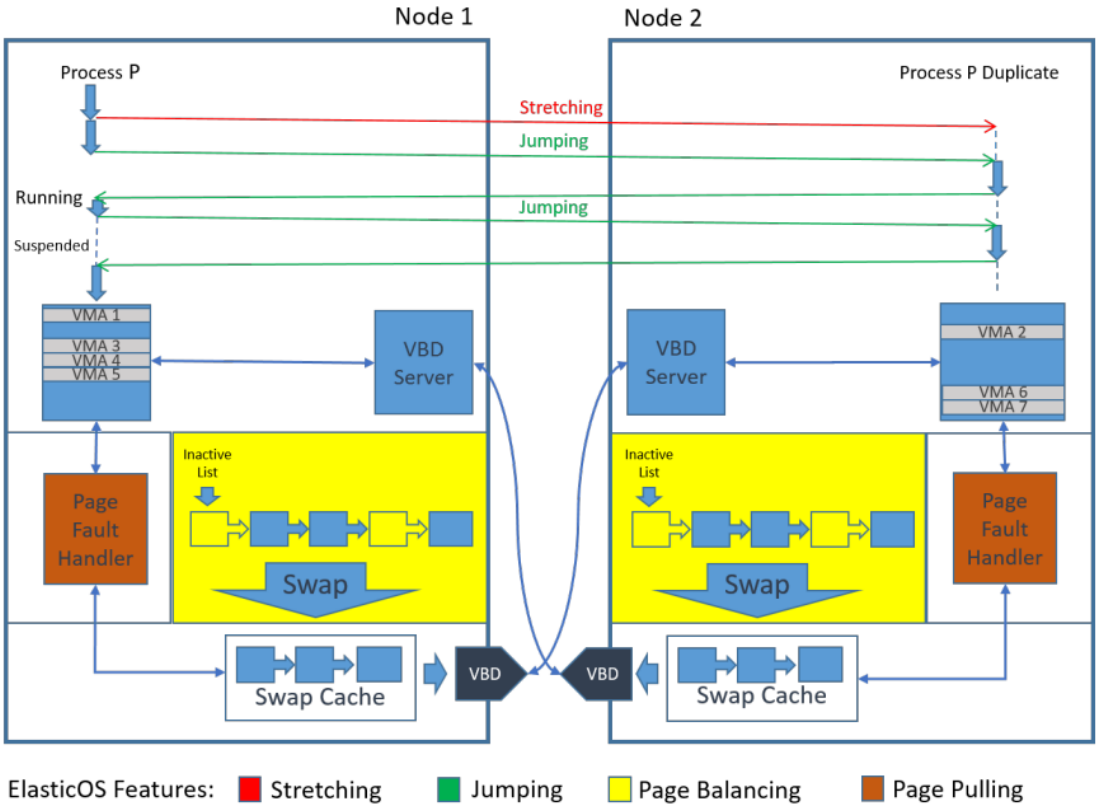


Figure 2.5: ElasticOS Architecture Design [1]

Page **Pull** primitive is used to copy data of the target process from remote node on demand. As shown in Fig.2.5, when the process triggers a page fault for a page in remote node, the page is then pulled from remote node to be accessed. Page **Push** shown as `Page Balancing` is used to perform network swap when the node is under memory pressure, where pushed pages are evicted to the remote node and available for access when a process Jumps to remote node. Page **Pull/Push** relies on Nswap[34] which implements a TCP network block device that drives the communication between nodes.

The agile execution transfer(**Jump**) capability of ElasticOS shows a significant performance improvement when compared to remote swap approach. The ability to stretch a process address space to multiple nodes and transfer execution in a lightweight manner unlocks savings in execution time as well as reduction in total network traffic. Due to the advantages and potential of ElasticOS primitives we will adopt the principals of `Stretch, Jump, Push and Pull` for Elastic Process framework.

Chapter 3

Design and Implementation

In this chapter we present our Elastic Process implementation and design. We adopt ElasticOS[1] principles and deliver the functionalities: Stretch, Jump, Pull and Push. In this chapter we will go over the details of Elastic Process framework components and highlight how our implementation and design differs from ElasticOS.

3.1 Motivation

Our design and implementation goal is to deliver fine-grained control functionality over processes while limiting the amount of changes to Linux kernel source code. Our implementation requires a very small change to the Linux kernel source code(15 lines of code total), where these changes only expose some of the kernel functions to our kernel modules to use and does not introduce or modify any of the default kernel functionalities as explained in 3.4.

We studied CRIU[12], ElasticOS[1] and LegoOS[40] to come up with an architecture design for Elastic Process that overcomes some of the limitations and challenges of CRIU, ElasticOS and LegoOS. The design and implementation of CRIU runs completely in userspace as shown by Fig.3.1. while userspace programs are easier to develop and debug, it is limited by the userspace capabilities. Userspace applications cannot access or manipulate the primitive data structures of processes, limiting how much control you can have over any application. On the other hand, ElasticOS and LegoOS live completely in kernel space, which gives you full control over processes. However, code development and

debugging is a tedious process. Also, modifying the operating system can result in many issues, including:

- stability problems: since the kernel is a complex system, and any changes not approved by the open source community can result in unpredictable behaviour.
- compatibility issue: modifying architecture specific code locks the system on a specific hardware. ElasticOS and LegoOS only run on x86 and requires a specific set of hardware and device drivers.
- upgradability issue: modifying a specific version of the OS source code means you cannot use it on later releases unless you modify the new release again.

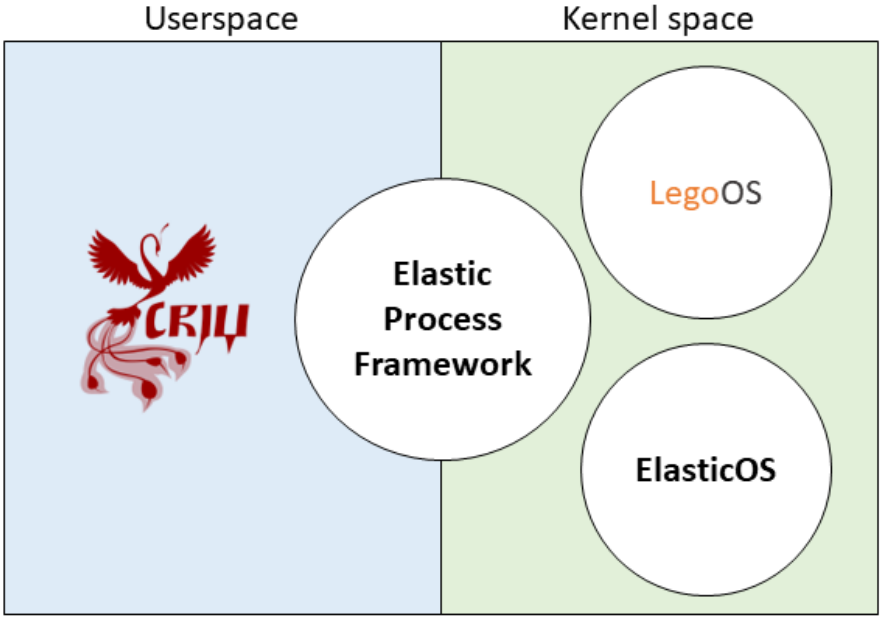


Figure 3.1: Performing Stretch on Elastic Process

Elastic Process implementation relies on loadable kernel modules for code running in kernel space, and for userspace we use CRIU and Ptrace[19]. As shown in Table.3.1 our design goals of minimizing modifications to the Linux kernel and implementing functionality in userspace aim to deliver joint disaggregation of memory and compute capabilities while overcoming the limitations of current systems. Our implementation demonstrates better stability, robustness and faster development time when compared to ElasticOS.

As shown in Table.3.1 above, Elastic Process delivers joint disaggregation of memory and compute without being bound and restricted by CPU architecture, special hardware,

System	Linux Kernel	Cross Platform	Agile Execution Transfer
ElasticOS	2.6	✗	✓
LegoOS	4.9	✗	✗
CRIU	3.11+	✓	✗
Elastic Process	5+	✓	✓

Table 3.1: Joint disaggregation of memory and compute systems comparison

and a specific Linux kernel release when compared to ElasticOS and LegoOS. While CRIU does not have hardware or Linux release restrictions, it does not deliver an execution transfer capability.

3.2 Linux Tools

Elastic Process framework utilize CRIU[12] and the debugger interface `Ptrace` to perform various operations on a target process, as well as manipulating the process state and address space. In this section, we explain how we integrate CRIU and Ptrace in Elastic Process framework.

3.2.1 CRIU

Check Point Restore in Userspace(CRIU) is a Linux tool used to freeze a running process, checkpoint the process state to desk, and copy the state to another machine where the process can be recreated and restored, and finally resume execution on the target machine. CRIU depends on **/proc** file system to read and dump a process state for the checkpoint procedure. For the restore procedure CRIU uses **fork()** system call to create a new process, then the newly created process is morphed into the target process.

One of the common use cases for CRIU is Lazy Migration, where CRIU checkpoint a process, migrate and resume execution immediately on the target node without waiting on the entire address space to be copied, then a post-copy of memory is performed.

We modify CRIU to use the lazy migration feature to implement **Stretch** functionality(see section 3.3.1). We use CRIU to clone the target Elastic Process on remote node without using CRIU's post-copy of memory, where we implement our own kernel modules to handle copying memory pages on demand(see section 3.3.3).

3.2.2 Ptrace

Ptrace is a Linux system call used to control, inspect and manipulate a target process. Ptrace is used in code analysis tools like gdb[42] and strace[13] to help developers with code development and debugging.

Ptrace attaches itself (tracer) to the target process(tracee) to gain control, examine and modify the tracee. Ptrace provides commands like PTRACE_ATTACH and PTRACE_DETACH to seize control and release a tracee respectively. Commands like PTRACE_GETREGSET and PTRACE_SETTREGSET are used to examine and modify the registers of the tracee respectively. Other commads like PTRACE_PEEKDATA and PTRACE_POKEDATA are used to read/write data from/to the tracee respectively.

We use Ptrace to implement the **Jump** functionality(see section 3.3.2. We use the commands mentioned above to control and manipulate the state of the Elastic Process, where we can update the registers and the stack of Elastic Process to provide a lightweight execution transfer between different machines.

3.3 Architecture Design

Our implementation and design goals are to deliver the following functionalities:

- Stretch: setting up and cloning Elastic Process on a remote machine for future execution transfer needs. Stretching a process results in expanding the address space to multiple nodes with synchronized memory mappings.

- Jump: lightweight execution transfer of Elastic Process state to a remote node by copying the minimal state required to resume execution on the target node.

- Pull: copy and swap in memory page from remote node after a page fault is triggered by Elastic Process.

- Push: swap out a page from memory of Elastic Process address space to a remote node due to memory pressure.

- Address space synchronization: a synchronization mechanism to ensures that all copies of Elastic Process are aware of any dynamic change to the address space.

3.3.1 Stretch

With Stretch we are spanning address space of Elastic Process on multiple nodes. We use the CRIU tool to implement the **Stretch** function. As shown in Fig. 3.2, we select Process A on Node1 as a target process to be elasticized, and our target machine will be Node2. We implemented "Elastic Process Daemon" shown as EP Daemon in Fig. 3.2. Our daemon is responsible for handling all commands sent to the elasticized process, and communicating with remote nodes for process state migration(see Section. 3.4). The following actions must be performed in order to Stretch Process A on Node2:

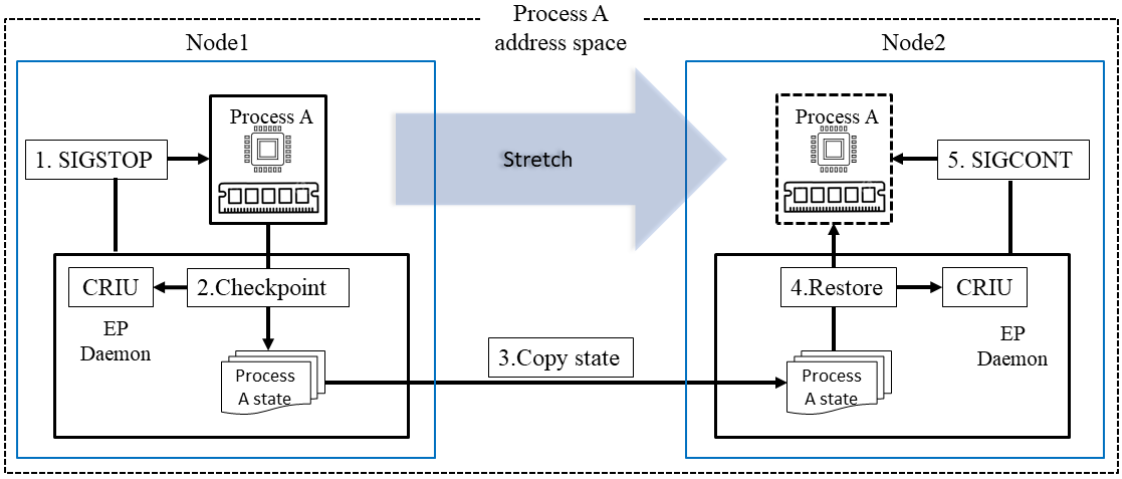


Figure 3.2: Performing Stretch on Elastic Process

1. SIGSTOP: sending SIGSTOP to Process A to force a context switch in the kernel, to allow the next step Checkpoint to read the target process state.

2. Checkpoint: EP daemon runs CRIU to perform checkpoint. CRIU uses the /proc file system in Linux to get all the information needed about the target process to be mirrored later in the restore stage. After gathering information about the target process, we write the collected data on a temporary file storage (tmpfs) for performance purposes. (See step 3)

3. Copy state: in this step we perform a desk-less migration, we pre-mount tmpfs, a memory mapped temporary file, on both Node1 and Node2 to read and write target process information. Then EP daemon runs secure copy(scp) to copy the process state information to the target node from our mounted tmpfs avoiding desk latency.

4. Restore: once all the process state information is copied to the target node's tmpfs, EP daemon runs CRIU again to create a replica of Process A on Node2. CRIU uses clone() system call to create a child process, then the child performs all the necessary actions to morph into Process A. The child process invokes system calls like mmap(), chdir() and chroot() to replicate memory maps, timers, credentials and any other state Process A may have.

5. SIGCONT: finally after the restore stage, the process is ready to resume execution. We send SIGCONT to signal to the kernel that the process is ready to be scheduled and run.

3.3.2 Jump

To provide a lightweight execution transfer we copy the minimum amount of state information for the target process to a remote machine and resume execution. We copy the CPU registers and the top two pages of the stack using Ptrace. In order for an Elastic Process to perform a Jump, we assume a Stretch has been done previously, since Jump assumes the target process already exists.

Fig. 3.3 shows an example of a Jump performed on Process A, where we copy the state of Process A from Node1 to Node2. Note Process A already exist on Node2 in a stopped state. Both nodes have an instance of Elastic Process Daemon(EP Daemon).

The following steps are performed to jump execution of Process A from Node1 to Node2:

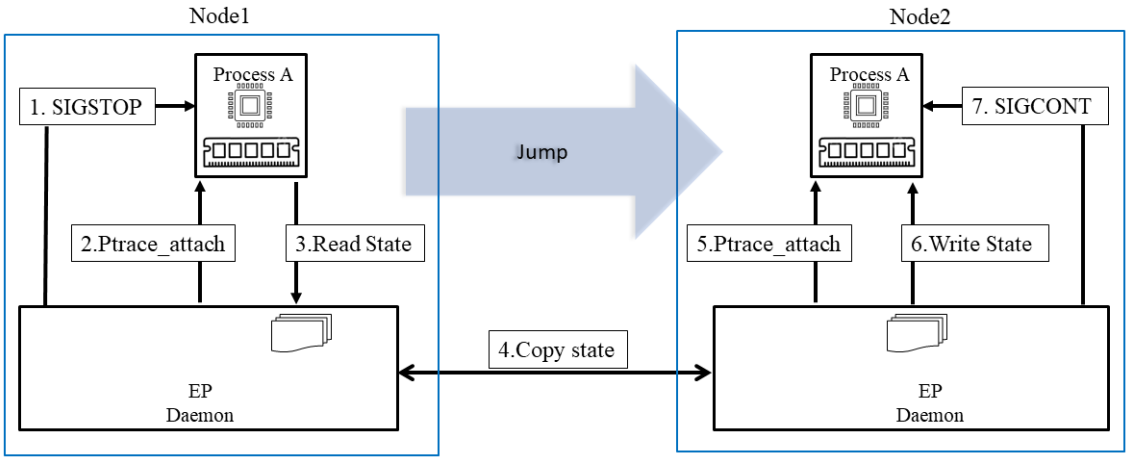


Figure 3.3: Performing Jump on Elastic Process

1. SIGSTOP: sending SIGSTOP to Process A to force a context switch in the kernel. A context switch is required to make sure the kernel dumps all the register values in struct user_regs_struct used later on step 3.

2. Ptrace_attach: EP daemon will call Ptrace_attach() to seize control of Process A.

3. Read state: read the CPU registers and stack using Ptrace PTRACE_GETREGS and PTRACE_PEEKDATA commands. The stack address is obtained from reading the register RSP.

4. Copy state: EP daemon sends the registers' values with the top two pages of the stack to Node2.

5. Ptrace_attach: size control of Process A on Node2, hence, Process A must exist on target node to perform jump.

6. Write state: write the CPU registers and stack to Process A using Ptrace commands PTRACE_SETREGS and PTRACE_POKEDATA commands.

7. SIGCONT: send SIGCONT to signal to the kernel that the process is ready to be scheduled and resume execution.

3.3.3 Page Pull

Our Page Pull/Push functionalities are implemented in a loadable Linux kernel modules and use a TCP client/server sockets that run in kernel space to transfer memory pages between machines.

Page Pull is triggered when Elastic Process tries to access a page that is not in local memory resulting in a page fault.

As shown in Fig. 3.4 Process A is running on Node1 and stretched to Node2. Note that page table entries for Process A indicate whether a page resides in local memory or in remote memory(address space synchronization is explained in 3.3.5). When Process A tries to access a page that resides in Node2 the following steps are performed for a page Pull:

1. Process A triggers a page fault for accessing a page that is not in local memory.

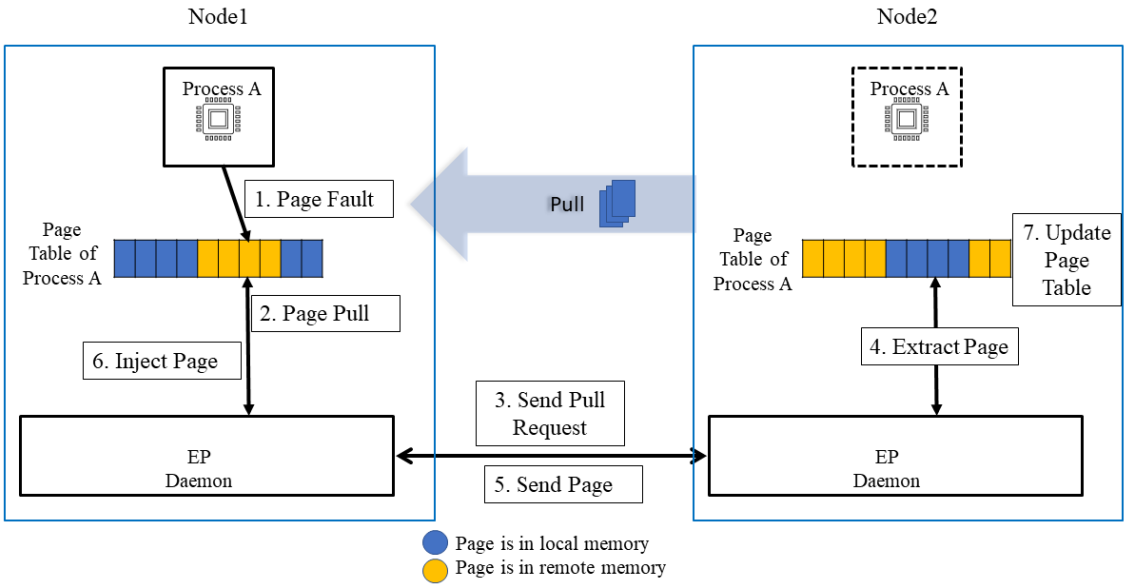


Figure 3.4: Page Pull

2. EP Daemon detects the page fault and creates a page Pull request.

3. A page Pull request is sent to EP Daemon instance running on Node2 using TCP client/server sockets running in kernel space.

4. On Node2, EP Daemon finds and extract the page from Node2's memory.

5. EP Daemon on Node2 responds with the requested page.

6. Page is injected into Node1's memory.

7. Page table entry on Node1 is updated to indicate that the page now resides in local memory.

8. Page table entry on Node2 is updated to indicate that the page now resides in remote memory.

3.3.4 Page Push

Page Push is triggered when a node running Elastic Process is running out of memory, which results in a spike of swapping activity. Linux kernel swap daemon selects candidate pages to swap out of memory in a least recently used(LRU) fashion. EP Daemon detects Elastic Process's pages being swapped out by the kernel swap daemon and

reroutes them to a node where Elastic Process have previously stretched to instead of the default swap space. Fig. 3.5 below shows Process A currently running on Node1 and

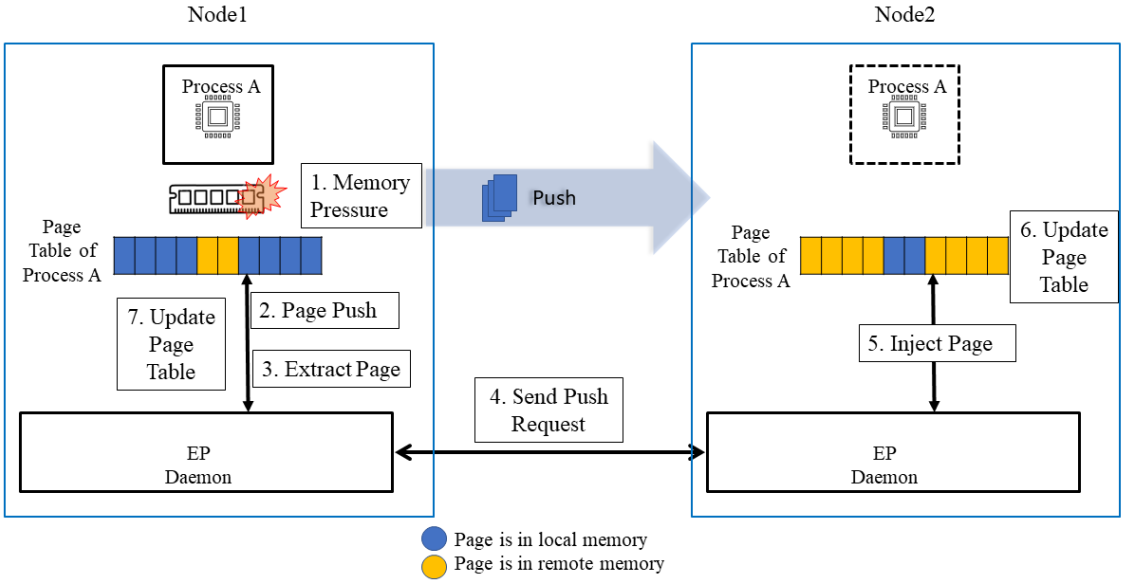


Figure 3.5: Page Push

experiencing memory pressure. The following steps are performed to push a page to a remote node:

1. Node1 is running out of memory and the kernel swap daemon is trying to swap out a memory page that belongs to Process A.

2. EP Daemon detects swapping activity and creates a page Push request.

3. Page is extracted from Node1's memory and added to the page Push request.

4. Page Push request is sent to Node2 with available free memory.

5. EP Daemon inject the page in memory.

6. Page table of Process A on Node2 is updated to indicate that the page is now in Node2's local memory.

7. Page table of Process A on Node1 is updated to indicate that the page is now in remote memory.

3.3.5 Address Space Synchronization

Address space synchronization is a loadable kernel module responsible for detecting any changes in Elastic Process page table. It essential to keep Elastic Process memory mappings synchronized across nodes to ensure that our Elastic Process have access to its entire address space at all times.

Our address space synchronization mechanism relies on mmap() and brk() system calls to detect and synchronize changes in Elastic Process page table. When mmap() is called, a new memory mapping is created for Elastic Process, and a sync update is sent to remote nodes to create the same new memory mapping. When brk() is called, the kernel updates updates the address of the end of data segment of Elastic Process, resulting in a sync update message sent to remote nodes to update their corresponding brk values for Elastic Process.

Fig 3.6 below shows how the page table entries are synchronized for Elastic Process.

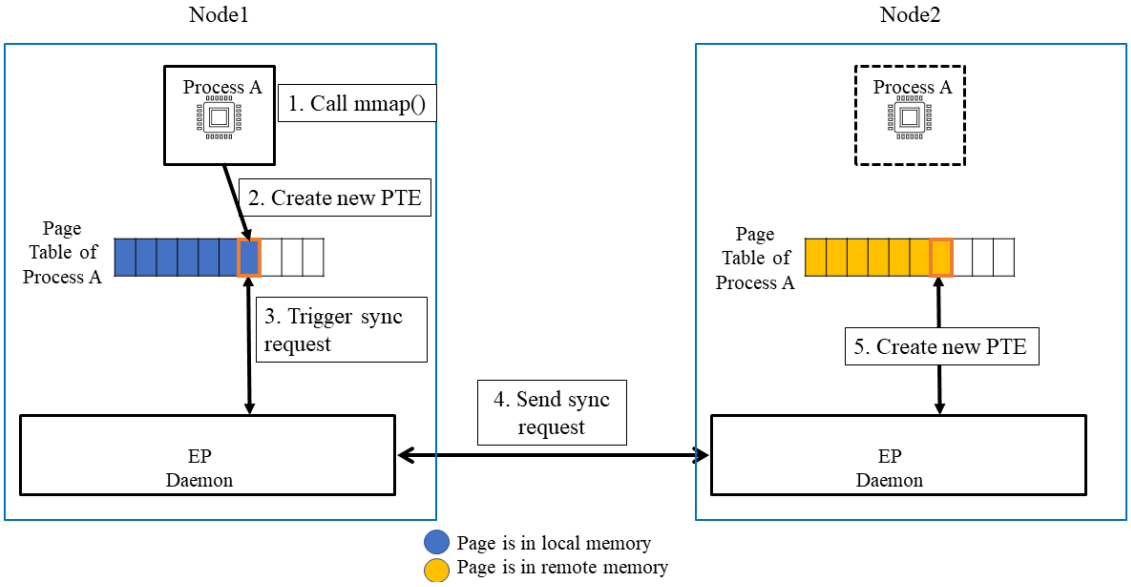


Figure 3.6: Address Space Synchronization

Following are the steps performed to synchronize the page table of Elastic Process: As shown in 3.6, we assume process A is running on Node1 and have been previously stretched to Node2.

1. Process A running on Node1 calls mmap() to creates a new memory mapping.

2. The call to mmap() results in creating a new page table entry(PTE) by the kernel.

3. Elastic Process Daemon detects the update in Elastic Process's page table, and creates a new sync request with the virtual memory address associated with the new PTE.

4. The sync request is then sent to Node2's Elastic Process Daemon.

5. Finally, Elastic Process Daemon picks up the new update and creates an identical PTE for Process A on Node2.

Linux kernel adopts a lazy approach when it comes to memory allocation. When a process creates a new memory mapping, the kernel associates the new PTE with a Zero Page physical memory. A Zero Page in Linux kernel is a physical page initialized with zeros and all new memory mappings point to this page. When a process tries to write to the new memory mapping, a page fault is triggered and only then the kernel will create a new page and associate it with a new physical address.

When Elastic Process with new memory mapping thatt is pointing to a zero page tries to write to this address, a page fault is triggered and a new physical memory is allocated. Elastic Process Daemon will detect the zero page fault and send a sync request to remote node to update the PTE to indicate that this page is now physically on remote node.

3.3.6 Multithreading

We created a proof of concept to support multi-threaded applications by adding a dispatcher functionality to Elastic Process Daemon. The creation of new threads in Linux kernel is very similar to creating a child process, the kernel calls `fork()` with `CLONE_VM` flag enabled to indicate that the memory is shared between. We have exposed `fork()` functionality to detect the creation of new threads in Elastic Process.

Supporting threads in Elastic Process Framework comes with many challenges including shared memory management and fine grained control over threads. As shown in Fig.3.7 we pin any shared memory between threads on one node referred to as Home Node, and any running threads on remote nodes can only access shared memory by jumping to

Home Node. We pin any shared memory to avoid the complexity of distributed locking implementation, which is a challenge we address in chapter 5.

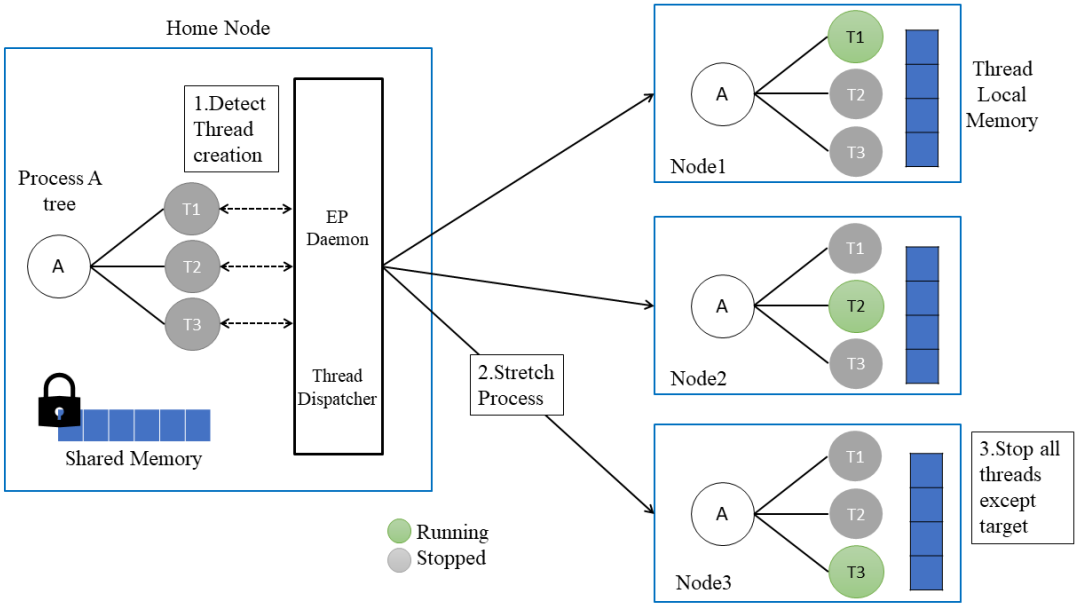


Figure 3.7: Elastic Process Daemon thread dispatcher

Fig.3.7 shows an example of our thread dispatcher design. EP Daemon is installed and running on Home Node as well as Nodes 1-3. The thread dispatcher module in EP Daemon is aware of three available nodes for scaling. When Elastic Process creates a new thread, the following steps are performed by Elastic Process Daemon:

1. EP Daemon detects the creation of a new thread `T3` by Process A via exposed `fork()` system call.

2. Thread dispatcher checks for available free nodes, and sends the Stretch request to Node3.

3. On Node3, Process A tree is cloned using CRIU, but non of the address space is copied.

    - To prevent T1 and T2 from running we attach Ptrace to to both threads, and leave our target thread T3 free to run.
    - Then Process A is able to continue execution on Node3.
    - Any thread local memory will be using Node3's physical memory.

- If T3 needs to access shared memory, T3 has to Jump to Home Node.

3.4 Implementation

Elastic Process Framework is implemented using C language and consists of two sets of programs:1)loadable kernel modules and 2)userspace programs. We also have a small modification to the Linux 5.13 kernel source code that exposes some of the kernel functionality to our kernel modules. Both userspace and kernel space programs work together and communicate using Linux signals to deliver the functionality of Elastic Process Framework.

3.4.1 Stretch

The goal of Stretch functionality is to have a clone with the minimum state of the Elastic Process on more than one node, to enable a lightweight transfer of execution between nodes. Our implementation of the Stretch functionality runs completely in userspace. We modify Checkpoint Restore in Userspace(CRIU) to dump the state of the target process to be elasticized. CRIU dump functionality typically copies the process including the entire address space. We utilize the lazy-migration functionality to implement the Stretch functionality.

In lazy-migration, CRIU copies the minimum state of the target process to start running on a different node, and lazily copies the rest of the process address space while the process is running. We enable the copy of minimum state required to run the target process using CRIU's lazy migration, however, we disable the post copy of the process address space and implement our own on demand paging.(see sections 3.4.3 and 3.4.4)

We use the following commands to Stretch process `EP` running on Node1 to Node2: Using modified CRIU tool to read the state of process `EP`, we run the following command on Node1:

```
sudo criu dump --images-dir mydir --shell-job --tree $EP
```

- `dump` command is used to read the state of a process and save it.
- `--image-dir mydir` to specify where the dumped process state is saved. `mydir` is a

memory mapped file system mounted using Linux `tmpfs`. We use `tmpfs` to improve the performance of reading and writing the process state.

- `--shell-job` used to indicate that we will attach the process to a shell on the remote node to see any standard output from the target process on terminal.
- `--tree $EP` is to specify the process id of our target, where `EP` is a variable holding the Elastic Process id.

After the process state have been saved, we copy the state files to Node2 using secure copy tool `scp`. On Node2 we run the following command to create a clone of our Elastic Process:

```
sudo criu restore --shell-job --images-dir mydir --leave-stopped
```

- `restore` command is used to create a process and restore the state of our copied Elastic Process.
- `--image-dir mydir` to specify where the state files are saved. `mydir` is also a `tmpfs` memory mapped file system.
- `--shell-job` used to indicate that we will attach the process to a shell on the remote node to see any standard output from the target process on terminal.
- `--leave-stopped` is to specify that we want to leave the process stopped after its creation since we will be transferring execution later using Jump.

Now we have a clone (excluding heap memory) of our Elastic Process on Node2 ready for execution transfer.

3.4.2 Jump

We use Ptrace to read the state of our Elastic Process. We implemented a TCP client/server program to transfer process state between nodes. The implementation of Jumping in Elastic Process Framework runs completely in userspace.

We implemented the following data structure to hold the state of Elastic Process to send/receive using our Jump TCP client/server:

```
struct ep_state{
    int pid;
    struct user_regs_struct u_regs;
    char *stack;
};
```

- `pid`: holds the process id of or target Elastic Process.
- `u_regs` is a `user_regs_struct` data structure that contains all CPU registers' values of the target process, including the instruction pointer register and stack pointer register.
- `stack`: is a pointer to a dynamically allocated memory that will hold the stack. Our current implementation assumes a stack size of 8KB.

Our Elastic Process Framework implementation triggers Jumping when Elastic Process pulls a number of remote pages that exceeds a preset `Threshold` value. A page Pull kernel module keeps track of how many pages have been pulled, and when the `Threshold` value have been reached, it resets the `Threshold` and sends `SIG_USR1` to the Jump client. Our client implements a signal handler which starts the Jumping process upon receiving `SIG_USR1`.

The following C code segment shows how Ptrace is used to read the state of Elastic Process:

```
 read_process_state(){
                    :
                    :
    ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    ptrace(PTRACE_GETREGS, pid, NULL, u_regs)
    stack_pointer = u_regs.rsp;
    read_stack(stack_pointer,pid);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);
                    :
                    :
```

```
}
```

- `PTRACE_ATTACH`: attach command is used by Ptrace to seize control of the target process with process id `pid`. Once Ptrace(tracer) attaches itself to target process(tracee), the tracee stops running and Ptrace can start examining and modifying the tracee.
- `PTRACE_GETREGS`: get registers command is used to read the CPU registers of the tracee, and save them in `u_regs`.
- `read_stack`: is a function responsible for reading the stack of the tracee.
  - `read_stack()`: calls `ptrace(PTRACE_PEEKDATA, pid, address, 0)` repeatedly untill 8KB of the stack is read.
  - `PTRACE_PEEKDATA`: command is used to read a word of data from `address` of the tracee.
- `stack_pointer`: is a 64-bit unsigned integer that holds the stack pointer of our tracee. The stack pointer is read from `user_regs_struct rsp` member.
- `PTRACE_DETACH`: finally Ptrace releases the tracee by calling a detach command.

Our TCP client handles reading the process state and sends an execution transfer request to the server on remote node along with the `ep_state` data. Before reading the process state, our TCP client sends `SIG_STOP` to the tracee to ensure it will remain stopped since we will be transferring execution to a remote node.

At the server side, when a Jump request is received with the corresponding `ep_state` data structure, the server will write the new state to the target process using the following code:

```
write_process_state(){
                    :
                    :
    ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    ptrace(PTRACE_SETREGS, pid, NULL, u_regs)
```

```
    stack_pointer = u_regs.rsp;
    write_stack(stack_pointer,pid);
    ptrace(PTRACE_DETACH, pid, NULL, NULL);
                    :
                    :
}
```

- `PTRACE_SETREGS`: set registers command is used to write `u_regs`to the registers of the tracee.
- `write_stack`: is a function responsible for writing the stack of the tracee.
  - `write_stack()`: calls `ptrace(PTRACE_POKEDATA, pid, address, data)` repeatedly untill 8KB of the stack is written.
  - `PTRACE_POKEDATA`: command is used to write a word of `data` into `address` of the tracee.

After the new state is successfully written, the server sends `SIG_CONT` to the target process to resume execution.

3.4.3 Page Pull

We implement page Pull and Push functionality in kernel modules to handle updates on Elastic Process page table as well as moving pages of memory between nodes. We implemented a kernel space TCP client/server to handle the address space synchronization.

When Elastic Process creates a new memory mapping it calls `mmap()` which we exposed in the Linux kernel to subsequently call a function in our kernel modules to send a page table update to remote machines. When `mmap()` is called by Elastic Process, the client will create and send a request to remote machine server containing information regarding the new memory mapping. On the server side, when a new memory mapping update is received, `mmap()` is called by the server on behalf of Elastic Process, which creates a replica of the memory mapping on remote node.

A similar process to `mmap()` update happens when Elastic Process updates the memory segment by calling `brk()` or releasing memory mappings by calling `munmap()`. Our

client will send the mapping update and on the server side `brk() or munmap()` is called on behalf of Elastic Process keeping memory mappings synchronized across nodes.

A page table entry(PTE) update is sent when Elastic Process allocates a new memory page. A memory allocation is triggered by Linux kernel page faulting mechanism. When a process tries to access a memory address for the first time a new PTE is created and added to the page table of that process. We exposed the Linux kernel `handle_pte_fault()` function to call an update function in our kernel module every time Elastic Process triggers a page fault.

Page Pull is triggered by issuing a page fault on a remote page. We handle the following page fault cases(not to be confused with Linux Major/Minor page faults, these are Elastic Process specific page faults):

1. Zero page: a zero page fault happens when a process issues a read access of a memory address for the first time. Linux kernel does not allocate physical memory until the process tries to write to the address, instead the kernel will create a new PTE that points to a kernel page initialized to zeros.

    - When zero page fault is triggered by Elastic Process, our client kernel module sends a PTE update to the server.
    - On the server side, we create a new zero page PTE and add it to Elastic Process page table using `set_pte_at()` function.

2. First time write fault: when a process tries to write to a memory address for the first time, the Linux kernel allocates a new page and updates PTE with the new physical address.

    - When a first-time write fault is triggered by Elastic Process, our client kernel module sends a PTE update to the server.
    - On the server side, we create a new PTE and add it to Elastic Process page table using `set_pte_at()` function. Then we set a special flag on the new PTE using `pte_mkspecial()` function, which indicates that the page is on remote node.

3. Major page fault: a major fault happens when a process tries to access a page that is not in memory. Typically the Linux kernel will look for that page in swap space.

    - When a major fault is triggered by Elastic Process, we check if the PTE special flag is set, which means the page lives in remote node.
    - The client issues a page Pull request and sends it to the server.
    - On the server side, when a page Pull request is received, the server will extract the page from memory and sends it back to the client, then the server will update PTE of the extracted page to indicate that its on remote node now by setting the special flag.
    - Once the request page is received at the client side, the page is injected in memory and the PTE associated with this page is updated with the new physical address, and we unset PTE special flag indicating that the page is now on local node.

3.4.4 Page Push

Page Push functionality is triggered by swapping activity in Linux kernel. When a machine is under memory pressure the Linux kernel swap daemon(`kswapd`) starts running and scans memory pages, `kswapd` creates a least recently used(`LRU`) list of pages to swap out. We expose `kswapd` function in Linux kernel and check if a page to be swapped out belongs to Elastic Process. When an Elastic Process page is a candidate for swap out, Elastic Process Framework runs the following procedure:

1. Detect `kswapd` attempt to swap out Elastic Process page and invoke page Push function running in the client kernel module.

2. The client extracts the page from memory, and sends a page push request to remote node.

    - We update PTE of the pushed page to indicate that it lives on remote node now.
    - We also release the page from `kswapd LRU` list and swap cache.

3. Once the server receives the Push request, it injects the page into memory and updates the PTE associated with that page to make it present in memory.

Our implementation of page Push mimics network swap approach, however, instead of swapping out to remote memory store, we systematically inject the swapped out page into the address space of Elastic process, which means the page can be accessed directly by the process after execution transfer(Jump) to the node that holds the page.

3.4.5 Linux 5.13 updates

In order for our Linux kernel modules for page `Push, Pull` and address space synchronization to work we need to expose some of the Linux kernel functions using `EXPORT_SYMBOL()` directive to make these functions available for kernel modules to use and not limited to the monolithic kernel use. We added the following lines of code to Linux 5.13:

- `mm/mmap.c EXPORT_SYMBOL(do_mmap);`

  This function is used by address space synchronization kernel module to create new memory mappings for Elastic Process. This call is triggered when a new memory mapping is created by one of Elastic Process clones.

- `mm/mmap.c EXPORT_SYMBOL(do_brk_flags);`

  This function is used by address space synchronization kernel module to move the break pointer(`brk`) for Elastic Process. This call is triggered when the brk pointer is updated by one of Elastic Process clones.

- `mm/mmap.c EXPORT_SYMBOL(do_munmap);` This function is used by address space synchronization kernel module to remove(unmap) a memory mapping of Elastic Process. This call is triggered when a memory mapping is removed by one of Elastic Process clones.

- `mm/swap.c EXPORT_SYMBOL(get_swap_page_of_type);` This function is used by address space synchronization kernel module as well as page Push and Pull kernel modules. This function is used to find a page table entry of a remote page, hence, remote pages are marked in the process page table as swapped pages.

In addition to exposing kernel functions we added the following code in `mm/memory.c` for the function `handle_pte_fault()` to help guide Elastic Process remote page faults:

```
if(current->elastic_flag && pte_remote(pte) && !pte_present(pte))
{
return handle_remote_fault(pte);
}
```

`handle_pte_fault()` function is used by the kernel to handle any page faults triggered by userspace programs. When Elastic Process triggers `handle_pte_fault()` the code above will redirect handling remote page faults to our kernel modules to Pull the remote page, following is an explanation of functions used:

- `current->elastic_flag`: used to check if the current process running is an Elastic Process. Whenever a new Elastic Process is created, we use our kernel modules to set a flag we added to `task_struct` to make it easier for us to track an Elastic Process in the kernel.
- `pte_remote(pte)`: check if the special bit is set for this page table entry(`pte`), which indicates that the page is in remote memory. When a page is swapped out, we set the special bit of the page's pte using the function `pte_mkspecial()`.
- `pte_present(pte)`: check if the page is present in memory.
- `handle_remote_fault(pte)`: once it is verified that the page is in remote memory, this function will call the page Pull kernel module to bring the page from remote memory and inject it into the current machine's memory.

3.4.6 Elastic threads

We implement a thread dispatcher that is triggered when an Elastic Process creates a new thread, a userspace script runs to Stretch the process tree to a new node using CRIU as mentioned in 3.4.1.

After stretching the process on a new node, the dispatcher function scans the process and creates parent process and a list of all of its threads using a userspace program, then we run the following code:

```
for each thread pid in pids except target thread:
ptrace(PTRACE_DETACH, pid, NULL, NULL);
```

We attach `Ptrace` to all threads except the target thread to prevent them from running. We cannot use the Linux kernel signal `SIGSTOP` because it is a special signal that stops the entire process(with all threads) and cannot be delivered to individual threads.

After `Ptrace` seizes control over Elastic Process threads we send `SIGCONT` to the process enabling only the target thread to run. When a running thread issues a Major page fault mentioned in 3.4.3, it means that the thread is trying to access a shared page on the Home Node. Instead of sending a page Pull request, we keep shared pages pinned and then we perform a Jump to the Home node instead.

When threads finish execution and are about to join the main process, we trigger a jump back to the Home node. This will ensure the aggregation of results on one node, where the home node will be responsible for collecting and reporting aggregated results.

Chapter 4

System Evaluation

In this chapter we evaluate Elastic Process Framework performance by comparing it to remote swap approach. We demonstrate the benefits of joint disaggregation of memory and compute with Jumping. Our results show performance improvement for memory bound applications due to reduced IO time of accessing remote pages when Jumping is enabled, while reducing the total amount of network traffic.

4.1 Experiments Setup

We ran all of our experiments on Chameleon Cloud[44]. We used bare metal Dell Power Edge R630 machines with the following hardware specifications:

- Architecture: x86_64
- Number of CPUs: 48
- RAM: 160GB

We use Ubuntu 20.04 with Linux 5.13. For experimental purposes we limit the machines available physical memory to be 10GB by modifying the boot parameters. We limit the available physical memory to create memory pressure scenarios for our performance evaluation.

For each experiment we set up two Chameleon bare metal nodes connected via 100Gbps Ethernet as shown below in Fig.4.1.

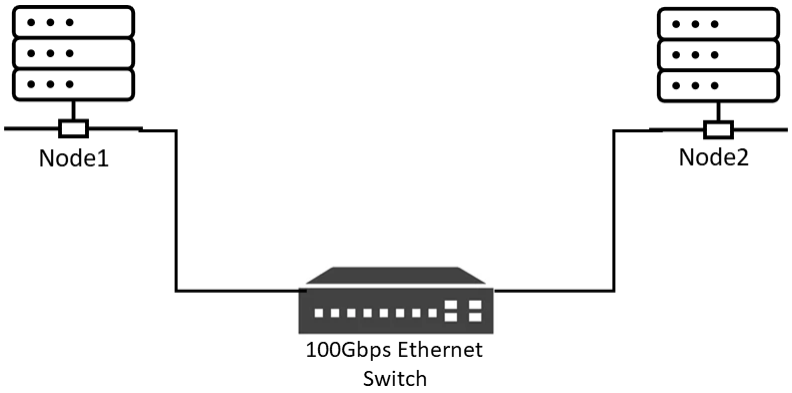


Figure 4.1: Experimental setup

We tested several applications on Elastic Process Framework with jumping and compared our results on the same framework with jumping disabled to create a remote swap scenarios.

4.2 Macro applications evaluation

We ran a total of 120 experiments and measured total run time and network traffic generated by 4 macro single threaded applications: 1)linear search 2)depth first search 3)count sort and 4)Dijkstra's algorithm. Each application has a total memory footprint of 16GB, and each node has 10GB of available memory.

We enabled jumping for Elastic Process using a simple jumping algorithm, where a jump is triggered when Elastic Process Pulls a number(Threshold) of remote pages. We ran several tests using a wide range of threshold values, where each application has a different threshold value in which it performs best. We offer more analysis on threshold values for jumping in 4.4.

Fig.4.2 below shows total runtime for each application compared to remote swap approach. Fig.4.3 shows the total network traffic generated by each of our macro applications compared to remote swap.

4.2.1 Linear search

Our linear search implementation creates a very large array of long integers(200 billion entries) and initialize the array with random positive integers, then the application

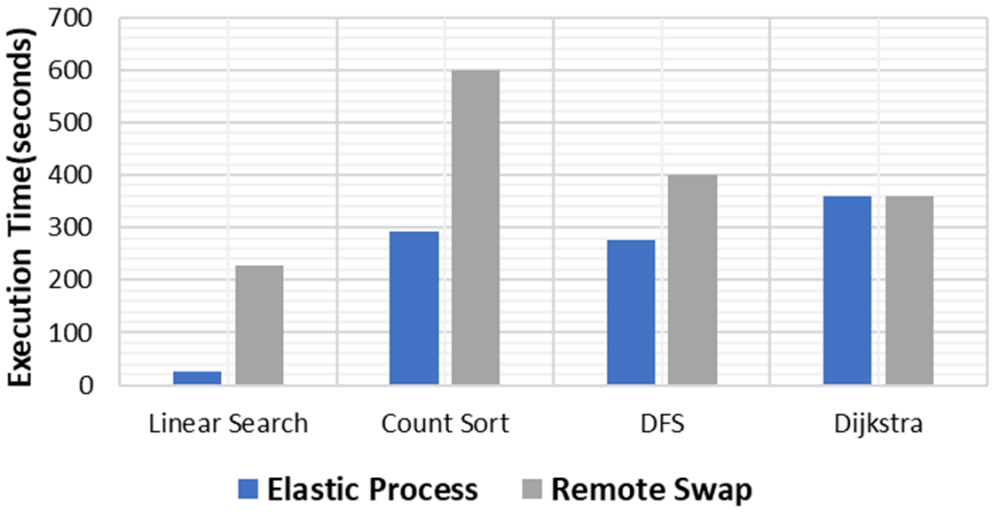


Figure 4.2: Execution time

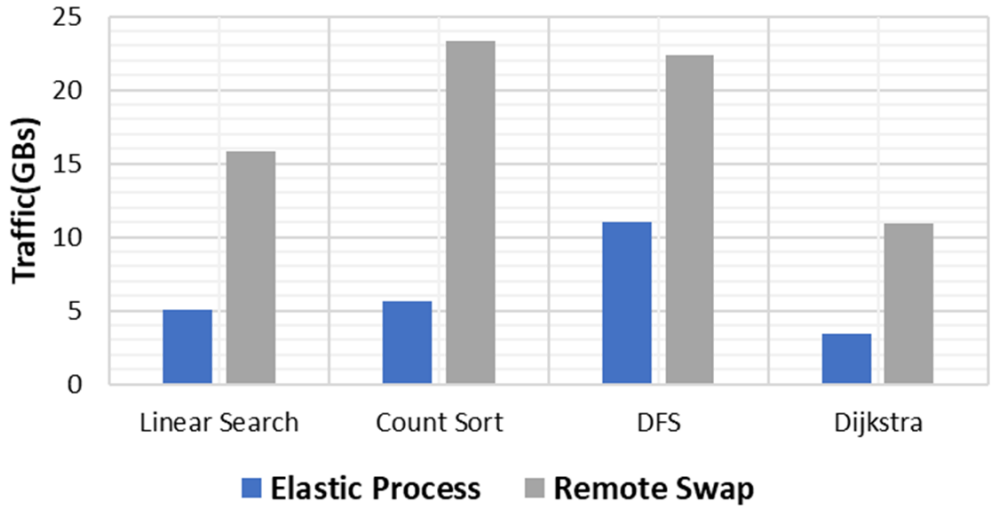


Figure 4.3: Network traffic generated

searches the entire array in a linear fashion. To ensure that our application touches on all memory pages we search for a value that does not exist in the array to avoid early termination.

Our experiments show that linear search performs nearly an order of magnitude faster on Elastic Process framework when compared to remote swap, while reducing the amount of network traffic by more than 3x.

4.2.2 Count sort

For count sort we create a large array of long integers(200 billion entries) for a total memory footprint of 16GBs. We initialize the array with random integers in the range(0-99). Then the application starts the count sort which uses an extra auxiliary space of 100 long integer entries to store the occurrence counts for each number.

Our experiments show that count sort experience more than 2x speedup in execution time compared to remote swap, while reducing the total amount of network traffic to nearly 5x.

4.2.3 Depth first search

We implement depth first search using C++. We create a class that contains `visited <int,bool>` map and `adj<int,int>` for adjacent nodes. We create a balanced tree of 166 million nodes for a total memory footprint of 16GB. We run a depth first search on the balanced tree until all nodes are visited.

Our test results show 1.4x speed up in execution time for depth first search when compared to remote swap, while decreasing the total amount of network traffic by more than 2x.

4.2.4 Dijkstra's algorithm

We use Dijkstra's algorithm to find the shortest path between all nodes and node 0 in a very large graph. In our implementation we create an integer matrix of size 60kx60k to hold the distances values. We initialize our matrix of distances to random values (1-10) then we start Dijkstra's shortest path search.

Our test results show no improvement on execution time when compared to remote swap. However, using Elastic Process framework decreased the total amount of network traffic generated by more than 2x. We do further investigation on performance behaviour in 4.4.

4.2.5 Elastic Process vs ElasticOS

We compare Elastic Process with ElasticOS using the same set of macro applications. We used the same setup for ElasticOS tests on machines with 10GB available physical memory and applications memory footprint of 16GB. We also used the same interconnect between nodes(100Gbps Ethernet) as shown in Fig.4.1.

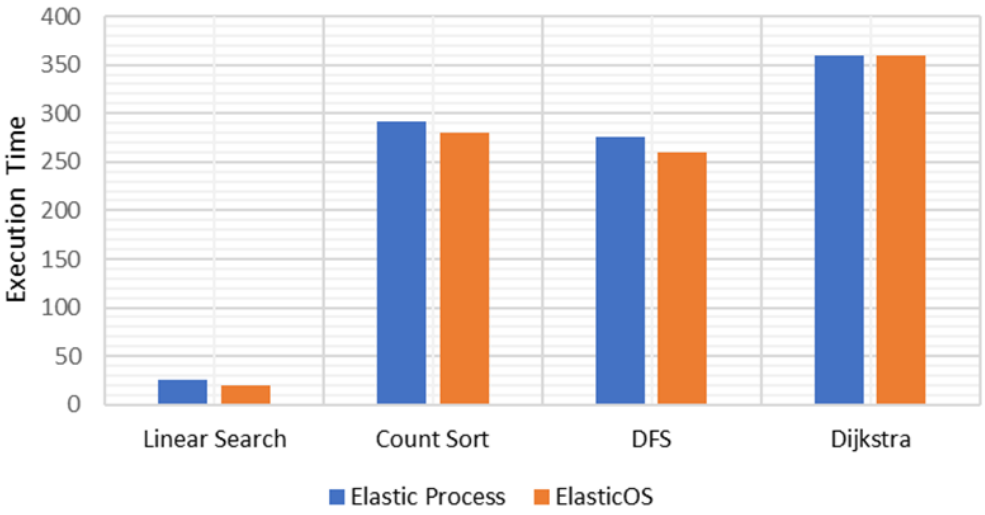
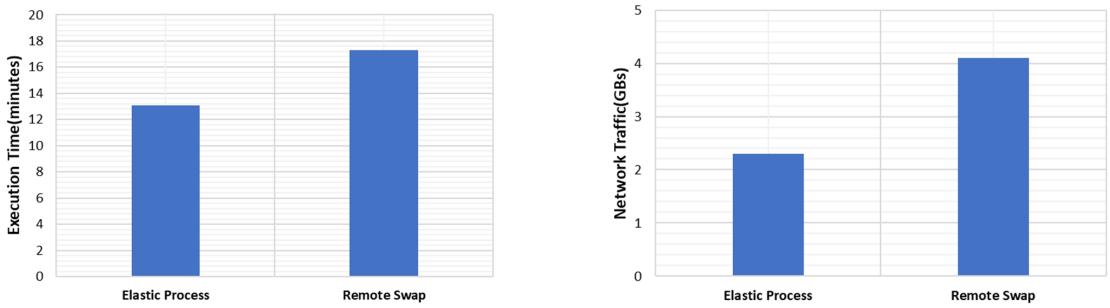


Figure 4.4: Execution time performance comparison of Elastic Process vs ElasticOS

Elastic Process shows a very similar performance to ElasticOS[1] with an average slow down of `3.5%` as shown in Fig.4.4. As explained in Chapter 3, Elastic Process shifts implementation to userspace, which results in a slight performance degradation when compared to ElasticOS.

Our design and userspace implementation improves the stability of the system when compared to ElasticOS. We ran a total of 100 experiments and measured the failure rate; failures including kernel crashes, kernel modules failure and userspace application failures. Our test results shows that ElasticOS tend to fail more than`50%` of the time, on the other hand, Elastic Process fails only `2%` of the time.

By shifting implementation to userspace Elastic Process framework is able to test off-the-shelf application as well as enabling multi-threaded support, where ElasticOS has limited support to single threaded macro application . The rest of our evaluation will not be compared to ElasticOS due to its limited ability to support off-the-shelf applications and multi-threaded applications.



(a) Execution time

(b) Network traffic

Figure 4.5: MySQL performance

4.3 MySQL evaluation results

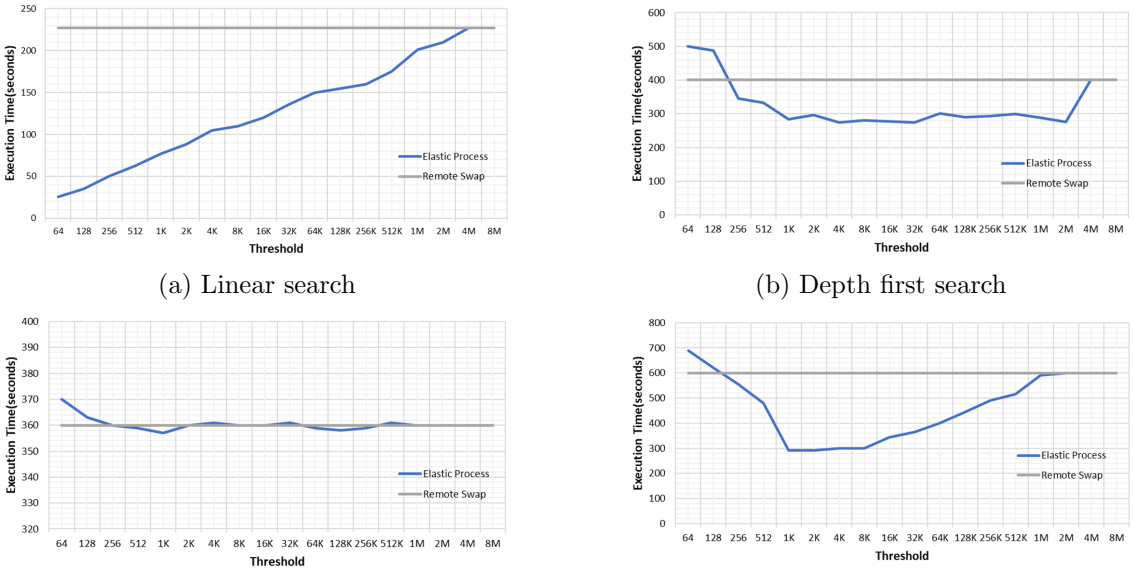
We also evaluate Elastic Process Framework for database table transformation using MySQL[32] . We populate a table with 200 million integer entries and use this mock up table to run our tests. Our experiment uses MySQL Memory Storage Engine[33] to load the entire table in memory and run MySQL `UPDATE` command, to modify all table entries, where the program's total memory footprint is 4GB.

Fig.4.5 above shows MySQL performance. As shown by Fig.4.5a, MySQL is 1.3X faster with jumping enabled, and Fig.4.5b shows a decrease in network traffic by 2x using Elastic Process compared to remote swap.

4.4 Jumping analysis

In this section we further investigate the application behaviour when jumping threshold value changes. We ran 18 experiments on each macro application with threshold value increasing exponentially from 64 to 8M, and we measured execution time for each Threshold value.

Fig.4.6 below shows how each application performs when we change the threshold value. We notice with very small threshold value, applications tend to perform worse than remote swap due to excessive jumping. When the threshold value is too small, a large amount of the application runtime is wasted on execution transfer back and forth between nodes, as well as slashing memory pages between nodes, resulting in worse performance when compared to remote swap.



(a) Linear search

(b) Depth first search

(c) Dijkstra's algorithm

(d) Count sort

Figure 4.6: Macro applications performance for different threshold values

On the other hand, when the threshold value is very large, Elastic Process performance becomes closer to remote swap until we reach a threshold value of 4M or higher. When the threshold value reaches 4M pages, it means the application has to pull 16GB of data before it jumps, and our test applications memory footprint is 16GB, which means the process never jumps and computation is pinned on one node, resulting in a performance similar to remote swap.

Linear search is the only application that performs best when threshold value is very small as shown by Fig.4.6a. Since linear search accesses the address space in a linear fashion, any pages swapped out to remote node are also in linear order, hence, Linux kernel swaps out pages in least recently used fashion. In other words, when linear search application starts to pull remote pages, it is most likely going to keep accessing pages that lives on remote node, therefore, an early Jump is recommended.

Applications like depth first search and count sort as shown by Fig.4.6b and Fig.4.6d respectively, shows a performance improvement when threshold value range between 128-1M. We noticed that the best threshold for depth first search and count sort is the minimum threshold value that yields the least runtime. Even though there are multiple threshold values that can have the same fastest runtime, the lower threshold value result in less network traffic, since the process will not Pull too many pages before a Jump is triggered.

Finally we further investigate Dijkstra's algorithm behaviour, where no significant improvement was shown for any threshold values. Dijkstra's algorithm test application is the only macro application that runs $O(n^2)$ complexity, making the application CPU bound, which means it spends most of the time in CPU rather than IO, so we measured CPU utilization for all macro applications and found a correlation with performance improvement.

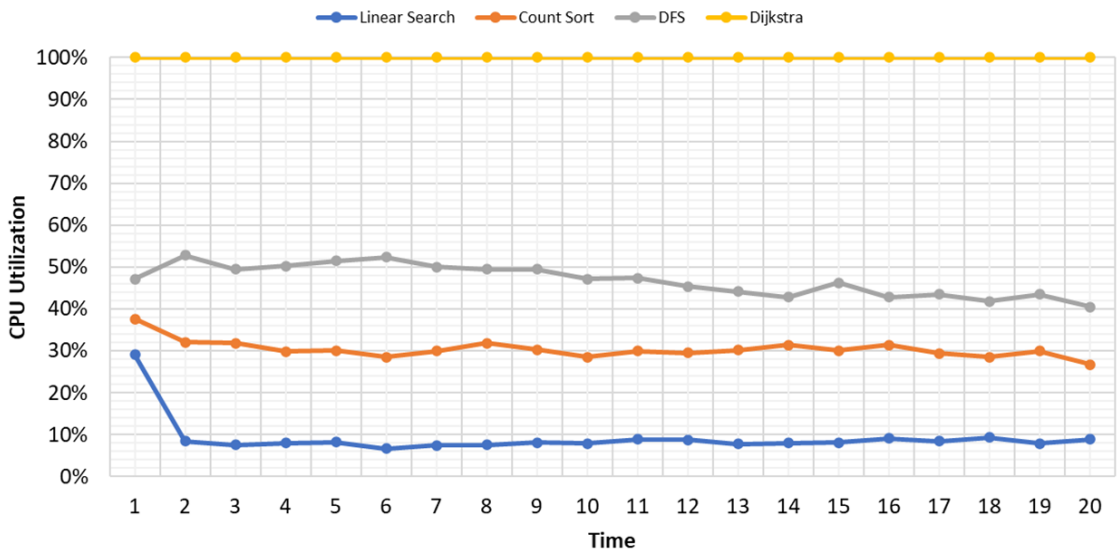


Figure 4.7: CPU utilization

Fig.4.7 above shows CPU utilization for our macro test applications. Since Elastic Process takes advantage of data locality by moving compute to data, Elastic Process reduces IO time since memory pages do not have to go through network when compute follows data. We noticed that linear search spends more time doing IO rather than CPU, thus, experiencing the most performance improvement from jumping. On the other hand, Dijkstra's spends most of the time doing computations and very little time doing IO, therefore, did not gain execution time performance improvement.

4.5 Multithreading evaluation

We created a multi-threaded linear search macro application to compare the performance of Elastic Process Framework single threaded with multi-threaded. We used the same single threaded linear search macro application from 4.2 with memory footprint of

16GB, and we created a multi-threaded version of the same application with two threads using C Posix Threads , each thread has a memory footprint of 8GB.

As shown in Fig.4.8 below we used 3 nodes to run the experiment, a Home Node to start the main process and aggregate the results, and two worker nodes to run one thread each(Node1 and Node2). All nodes are connected via 100Gbps Ethernet switch.

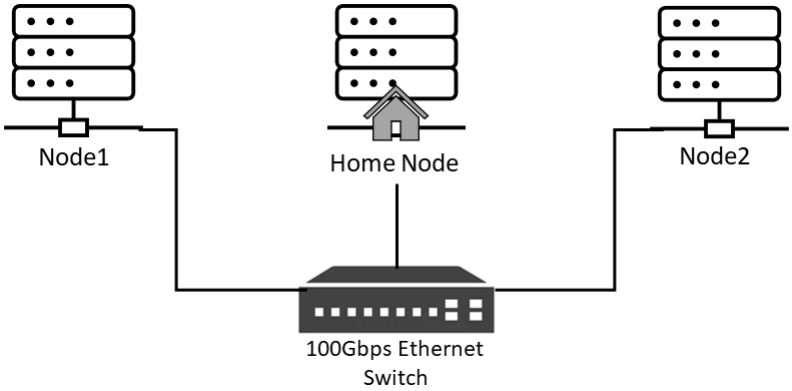


Figure 4.8: Multi-threaded experiment setup

In this experiment we pin any shared memory between the threads on **Home Node**. We also pin any thread local memory on Node1 and Node2. Before any thread is done executing it joins the main thread performing a Jump to the **Home Node**. The linear search results are aggregated by threads at the end of execution.

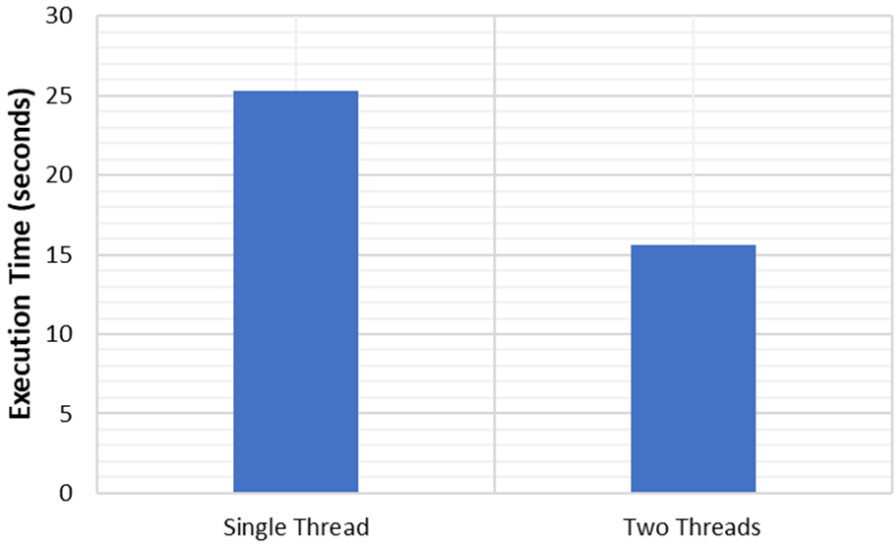


Figure 4.9: Elastic Process linear search performance

As shown in Fig.4.9 above, running two threads to search the same address space

results in 1.6x execution time improvement. Elastic Process framework was able to distribute threads automatically without requiring any changes to the program, effectively abstracting away the complexity of the infrastructure from the developer.

Chapter 5

Future Work

5.1 Application support

We plan to expand on Elastic Process support for more off-the-shelf applications. Our current implementation of Elastic Process Framework only supports desegregation for memory and compute, and in order to support more off-the-shelf applications we need to add support for IO and networking. We also plan to test and evaluate Elastic Process Framework by running machine learning and more database applications.

5.2 Jumping

Our initial results show that memory bound applications tend to benefit the most from jumping, however, we need to have a better understanding on what threshold value to use for different applications, and we need to implement an adaptive algorithm that changes the threshold value dynamically while the application is running.

We also intend to conduct a study on different applications memory access patterns and employ machine learning to help us understand and develop a more mature jumping policy that can be adaptive and make a more informed jumping decision.

5.3 Multithreading support

We intend to compare Elastic Process Framework to data processing frameworks like Spark[46] and Hadoop[3]. We will also leverage Elastic Process Jump, Pull and Push

mechanisms for Elastic Threads and allow threads to span across multiple nodes. Threads ability to utilize resources on remote machines can help solve problems with straggler worker threads for distributed processing.

We will explore distributed locking mechanisms and location aware paging to enable shared memory pages to be pulled/pushed by threads, which will enable Elastic Process to support more applications.

Chapter 6

Conclusion

In conclusion, Elastic Process Framework provides an auto scaling with joint disaggregation of memory and compute capabilities while overcoming the limitations of hardware and software requirements in comparison to ElasticOS, LegoOS and CRIU. Elastic Process Framework demonstrates an average of 2x execution time performance improvement and 60%-70% network traffic reduction when compared to remote swap approach. Our test results show that memory bound applications tend to reduce execution time in comparison to CPU bound applications.

The design and implementation of Elastic Process demonstrates stability and robustness due to the minimal changes to the Linux kernel, as well as moving development to userspace, which also improves development and debugging time.

Our test on linear search with multiple threads show that Elastic Process Framework is able to run multi threaded macro application without requiring any modification to the application. The automatic thread distribution mechanism abstracts away the complexity of the infrastructure from the developer.

Our parameter analysis shows that the performance of test applications is heavily impacted by jumping decision, and tuning the parameters can be improved by studying the memory access patterns of different applications and potentially using machine learning to create an adaptive jumping policy.

References

[1] Ehab Ababneh et al. "Elasticizing Linux via Joint Disaggregation of Memory and Computation". In: *arXiv preprint arXiv:1806.00885* (2018).

[2] Arwa Aldhalaan and Daniel A Menascé. "Analytic performance modeling and optimization of live VM migration". In: *European Workshop on Performance Engineering*. Springer. 2013, pp. 28–42.

[3] Apache Software Foundation. *Hadoop*. Version 0.20.2. Feb. 19, 2010. URL: `https://hadoop.apache.org`.

[4] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[5] Cristian Hernandez Benet, Kyoomars Alizadeh Noghani, and Andreas J Kassler. "Minimizing live VM migration downtime using OpenFlow based resiliency mechanisms". In: *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*. IEEE. 2016, pp. 27–32.

[6] Haakon Bryhni, Espen Klovning, and Oivind Kure. "A comparison of load balancing techniques for scalable web servers". In: *IEEE network* 14.4 (2000), pp. 58–64.

[7] Juan Cáceres et al. "Service Scalability Over the Cloud". In: *Handbook of Cloud Computing*. Ed. by Borko Furht and Armando Escalante. Boston, MA: Springer US, 2010, pp. 357–377. ISBN: 978-1-4419-6524-0. DOI: `10.1007/978-1-4419-6524-0_15`. URL: `https://doi.org/10.1007/978-1-4419-6524-0_15`.

[8] Blake Caldwell et al. "FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud". In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 665–677. DOI: `10.1109/ICDCS47774.2020.00090`.

[9] Emiliano Casalicchio. “Container orchestration: A survey”. In: *Systems Modeling: Methodologies and Tools* (2019), pp. 221–235.

[10] Paul Castro et al. “The rise of serverless computing”. In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[11] David Cohen et al. “Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options”. In: *2009 17th ieee symposium on high performance interconnects*. IEEE. 2009, pp. 123–130.

[12] CRIU Dec. *Checkpoint/Restore in Userspace*. 2015.

[13] Mathieu Desnoyers and Michel Dagenais. “OS tracing for hardware, driver and binary reverse engineering in Linux”. In: *CodeBreakers Journal* 1.2 (2006).

[14] Harjot Dhawan and Bineet Kumar Joshi. *Road to Cloud Computing*. Lap Lambert Academic Publ, 2012.

[15] Geoffrey C Fox et al. “Status of serverless computing and function-as-a-service (faas) in industry and research”. In: *arXiv preprint arXiv:1708.08028* (2017).

[16] Juncheng Gu et al. “Efficient memory disaggregation with infiniswap”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 649–667.

[17] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. “Post-copy live migration of virtual machines”. In: *ACM SIGOPS operating systems review* 43.3 (2009), pp. 14–26.

[18] Divya Kapil, Emmanuel S Pilli, and Ramesh C Joshi. “Live virtual machine migration techniques: Survey and research challenges”. In: *2013 3rd IEEE international advance computing conference (IACC)*. IEEE. 2013, pp. 963–969.

[19] Jim Keniston et al. “Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps”. In: *Proceedings of the 2007 Linux symposium*. 2007, pp. 215–224.

[20] Jeongchul Kim and Kyungyong Lee. “Practical cloud workloads for serverless faas”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 477–477.

[21] Mariam Kiran et al. “Lambda architecture for cost-effective batch and speed big data processing”. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 2785–2792.

[22] Agus Kurniawan and Wely Lau. “Introduction to Azure Functions”. In: *Practical Azure Functions*. Springer, 2019, pp. 1–21.

[23] Kevin Laubis, Viliam Simko, and Alexander Schuller. “Cloud Adoption by Fine-Grained Resource Adaptation: Price Determination of Diagonally Scalable IaaS”. In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2015, pp. 249–257.

[24] *LegoOS GitHub page*. URL: `https://github.com/WukLab/LegoOS#3-a-nameplatformrequirementap requirement`.

[25] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. “High performance RDMA-based MPI implementation over InfiniBand”. In: *International Journal of Parallel Programming* 32.3 (2004), pp. 167–198.

[26] Ling Liu et al. “Memory disaggregation: Research problems and opportunities”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 1664–1673.

[27] Fei Ma, Feng Liu, and Zhen Liu. “Live virtual machine migration based on improved pre-copy approach”. In: *2010 IEEE International Conference on Software Engineering and Service Sciences*. IEEE. 2010, pp. 230–233.

[28] Maciej Malawski et al. “Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions”. In: *Future Generation Computer Systems* 110 (2020), pp. 502–514.

[29] Garrett McGrath and Paul R Brenner. “Serverless computing: Design, implementation, and performance”. In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, pp. 405–410.

[30] Mohammadreza Mesbahi and Amir Masoud Rahmani. “Load balancing in cloud computing: a state of the art survey”. In: *International Journal of Modern Education and Computer Science* 8.3 (2016), p. 64.

[31] Victor Millnert and Johan Eker. "HoloScale: horizontal and vertical scaling of cloud resources". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020, pp. 196–205. DOI: 10.1109/UCC48980.2020.00038.

[32] *MySQL*. Dec. 22, 2022. URL: https://www.mysql.com/.

[33] *MySQL Memory Engine*. URL: https://dev.mysql.com/doc/refman/5.7/en/memory-storage-engine.html.

[34] Tia Newhall et al. "Nswap: A network swapping module for linux clusters". In: *European Conference on Parallel Processing*. Springer. 2003, pp. 1160–1169.

[35] Vlad Nitu et al. "Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter". In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–12.

[36] John Ousterhout et al. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015). ISSN: 0734-2071. DOI: 10.1145/2806887. URL: https://doi.org/10.1145/2806887.

[37] Carlo Puliafito et al. "Container migration in the fog: A performance evaluation". In: *Sensors* 19.7 (2019), p. 1488.

[38] Sasko Ristov et al. "Superlinear speedup in HPC systems: Why and when?" In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016, pp. 889–898.

[39] Josep Sampé et al. "Serverless data analytics in the ibm cloud". In: *Proceedings of the 19th International Middleware Conference Industry*. 2018, pp. 1–8.

[40] Yizhou Shan et al. "Legoos: A disseminated, distributed OS for hardware resource disaggregation". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 69–87.

[41] Gursharan Singh and Parminder Singh. "A Taxonomy and Survey on Container Migration Techniques in Cloud Computing". In: *Sustainable Development Through Engineering Innovations: Select Proceedings of SDEI 2020*. Springer Singapore. 2021, pp. 419–429.

[42] Richard Stallman, Roland Pesch, Stan Shebs, et al. “Debugging with GDB”. In: *Free Software Foundation* 675 (1988).

[43] Nikos Tziritas et al. “Online live VM migration algorithms to minimize total migration time and downtime”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 406–417.

[44] *Welcome to chameleon*. URL: `https://chameleoncloud.readthedocs.io/en/latest/index.html`.

[45] Lenar Yazdanov and Christof Fetzer. “Vertical Scaling for Prioritized VMs Provisioning”. In: *2012 Second International Conference on Cloud and Green Computing*. 2012, pp. 118–125. DOI: 10.1109/CGC.2012.108.

[46] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi.org/10.1145/2934664.

[47] Yingqiang Zhang et al. “Towards cost-effective and elastic cloud database deployment via memory disaggregation”. In: *Proceedings of the VLDB Endowment* 14.10 (2021), pp. 1900–1912.

[48] Albert Y Zomaya. *Advanced computer architecture and parallel processing*. 2021.