COL732 Project Report

Anirudha Kulkarni Sachin Ronak Ladhar
Sourav Sharma Chintan Sheth K Laxman

November 2022

1 Introduction

A snapshot allows one to pause a running VM and resume elsewhere. This is useful for taking backup, Live migration for load balancing, forking a VM from base image, and disaster recovery. The project aims to provide an efficient and scalable implementation of snapshotting API for main project of COL732. The project is implemented in rust for safety and performance. The API is used by Load Balancer for Live Migration of VMs, frontend for forking base image, and anti-cheating system for monitoring access to VMs. Slides presented can be found here

2 OKRs

The project was divided into 5 main stages:

- A crate to pause and resume
  - Barebone VM from KVM api
  - BZImage from Linux kernel
  - Generic Linux distribution (Ubuntu, Fedora, etc.)
- Adding the functionality to manually interrupt and pause running VM
- Converting Crate to a Microservice
- Webserver to Spawn multiple VMs
- An efficient way to store snapshot for multiple VMs - Deduplication

We achieved all of the OKRs before the deadline of 31st October 2022.

3 Design Assumptions

We make the following assumptions:

- KVM as the hypervisor
- VM can be any linux distribution (Ubuntu, CentOS, etc.)
- VM can have multiple CPUs, memory and devices (disks, network interfaces, disk etc.)
- x86_64 architecture

4 Architecture

The VMM has been developed in Rust on top of vmm-reference provided by Rust-VMM. The VMM manages the lifecycle of the VM from fork to halt and also emulates the devices to allow the VM to perform I/O and MMIO reads and writes. The KVM utility provided by Linux has been used as the base which provides the APIs to run and manage VMs.

4.1 tarpc - RPC between VMM and Web Server

RPC(Remote Procedure Calls) is a mechanism for communication between different processes. RPC allows communication across machines on the same network, but in our case we have used the mechanism for communication between processes on the same machine. Due to limitation of the vmm-reference hypervisor which allows to create only one VM, and it owns the object of the created VM for any further communication.

Since the webserver cannot directly transfer requests for taking snapshots to individual VMs, RPC was implemented between the webserver and VMM. All VMMs act as RPC servers, exposing a port where they listen for requests from a web server that acts as a client and transmits API requests for taking snapshots to the appropriate VMM.

Tarpc is a Rust-based RPC framework which allows us to define the RPC services on the server and provides the mechanism to call these services from the clients with much ease.

4.2 Rocket - Web Server for exposing API Endpoints

As the primary backend for the application has been developed in Python, in order to avoid the language dependencies, it was better to build a web server which can listen to requests from any client irrespective of its environment. Due to limitations of RPC, it had to be built in Rust, thus Rocket seemed to be the best choice for exposing standard API endpoints for communicating requests to appropriate VMs.

Rocket is a rust-based web framework that is quick, simple, and adaptable. Additionally, it offers safety and security guarantees whenever possible. Besides that, it allows the user to write as minimal code as possible.

4.2.1 Exposed APIs

- **Snapshot API (POST) Endpoint :** <server_base_url>/snapshot
  Parameters
  - **cpu_snapshot_path (String) :** path at which cpu snapshot will be stored
  - **memory_snapshot_path (String) :** path at which memory snapshot will be stored
  - **rpc_port (Integer) :** port corresponding to the VMM's RPC server which is running the desired VM
  - **resume (Bool) :**
    True if want to resume the VM after taking the snapshot.
    False if want to halt the VM after taking the snapshot

  Request

  Listing 1: Snapshot POST API Request

  ```
  {
     "cpu_snapshot_path" : "<cpu_path>",
     "memory_snapshot_path" : "<memory_path>",
     "rpc_port": <port>,
     "resume": <true/false>
  }
  ```

  Response

  Listing 2: Snapshot POST API Response

  ```
  {
      "Snapshot Taken Successfully"
  }
  ```

- **Restore or Create API (POST) Endpoint:** <server_base_url>/create
  Parameters
  - **cpu_snapshot_path (String) :** the path to saved CPU snapshot if want to resume from existing VM
  - **memory_snapshot_path (String) :** the path to saved memory snapshot if want to resume from existing VM
  - **kernel_path (String) :** the path to base kernel image if want to create fresh VM from given kernel image

- **resume (Bool) :**
  True if want to resume from existing VM snapshot
  False if want to create a fresh VM from base kernel image

Request

Listing 3: Restore or Create POST API Request

```
{
    "cpu_snapshot_path" : "<cpu_path>",
    "memory_snapshot_path" : "<memory_path>",
    "kernel_path" : "<kernel_image_path>",
    "resume": <true/false>
}
```

Response

Listing 4: Restore or Create POST API Response

```
{
    "pid" : <Process PID>,
    "port" : <RPC Port>
}
```

5 Workflow

Backend manages the mapping from base image name to its appropriate path on local storage or shared network storage. When instructor wants to setup a VM for the course assignment, he/she selects the base image name from the available options together with cpu and memory configurations. After filling up the desired configuration, clicks on create VM. Backend on receiving create VM request will translate it into POST request containing appropriate kernel_path, to the web server listening for API requests. On receiving create request, web server will create a new process of the "vmm" binary by providing the path to the kernel image and assigns it a unique port for the RPC server bound with the VMM which will be later used for communicating any further messages to the VMM of the corresponding VM. In response webserver will send assigned rpc port and pid of the VMM process to the backend. Instructor will install all the required packages, copies assignment related documents, starter code, etc on newly created VM. Once the VM is setup instructor selects snapshot and pause option to take live snapshot of the VM. Backend on receiving the snapshot request will send appropriate rpc port of VMM together with cpu and memory snapshot paths at which snapshot of the VM will be stored. When instructor wants to create VMs for the students enrolled in the course, he selects appropriate snapshot name from which the VMs needs to be created. Backend stores mapping from snapshot name to the cpu and memory snapshot paths.

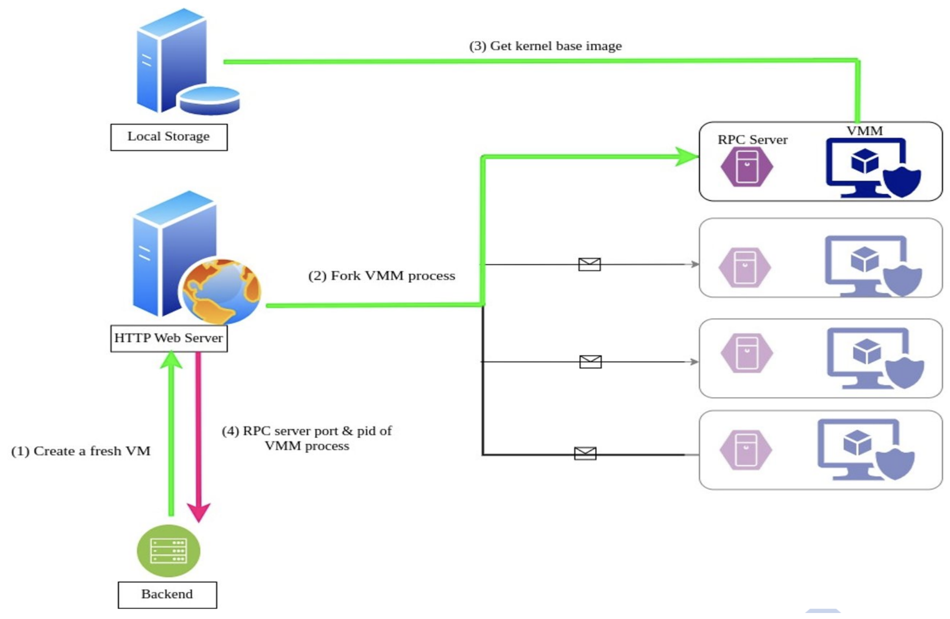

Figure 1: Flow for creating VM

Backend will send create API request to the webserver with appropriate cpu and memory snapshots paths along with resume flag set as true. On receiving create request, web server will start VM from stored cpu and memory snapshot along with rpc server listening on a randomly assigned port which will later be used for communicating any further requests to the VMM of corresponding VM. In response webserver will send assigned rpc port and pid of the VMM process to the backend.

Later when the instructor asks for the allocation of VMs providing the name of the snapshot saved and calls create API with resume flag as true. Backend will pass the location of the appropriate cpu_snapshot_path and memory_snapshot_path corresponding to the saved snapshot of instructor, to create API for each and every student in the batch. The VMM will fetch the saved memory and CPU states from the provided paths and resume the VM from the same state at which the snapshot was taken. Later when a snapshot is called, the backend will provide the location to save the cpu snapshot and memory snapshot (which will be unique for each VM and may be different from the previous path from which it is created) and we will simply save the snapshot to that location. In response, we will send the rpc_port on which VMM listens to the future commands from the web server as well as pid of the newly created VMM process. Backend will manage mapping from u_id and course_id to rpc_port and pid. In addition to managing rpc_port and pid, it also manages storage locations for cpu and memory states on a per VM basis. It also holds the base kernel image from which VM will be initially created for the instructor.

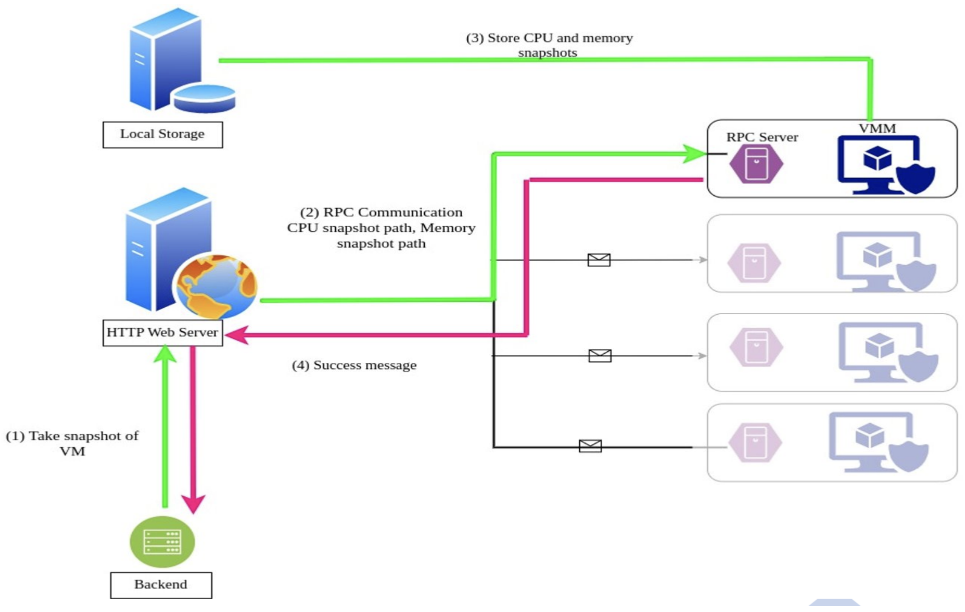
(3) Store CPU and memory snapshots
Local Storage
RPC Server
VMM
(2) RPC Communication CPU snapshot path, Memory snapshot path
HTTP Web Server
(4) Success message
(1) Take snapshot of VM
Backend

Figure 2: Flow for snapshot VM

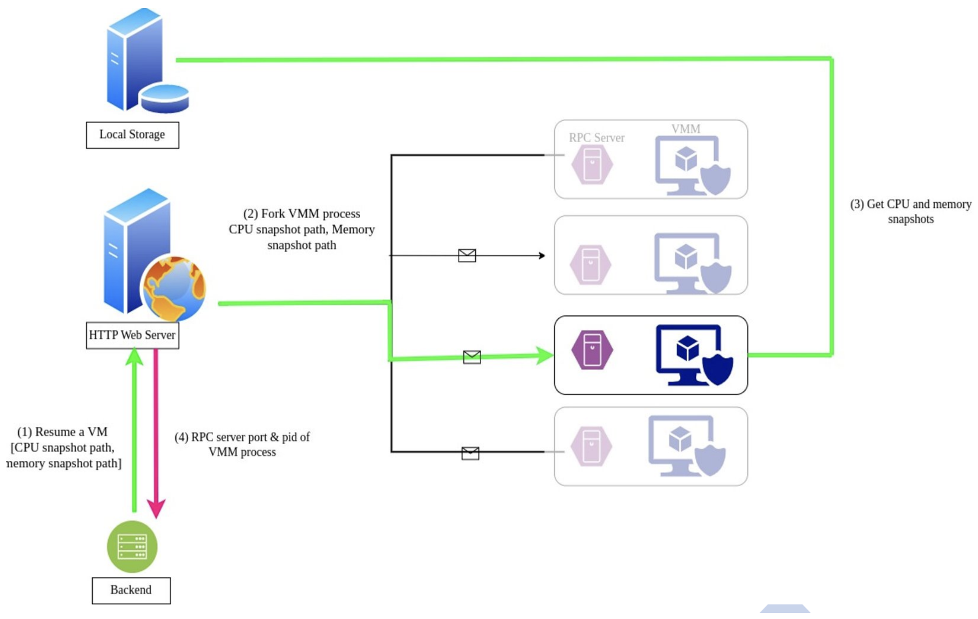
Local Storage
RPC Server
VMM
(2) Fork VMM process CPU snapshot path, Memory snapshot path
(3) Get CPU and memory snapshots
HTTP Web Server
(1) Resume a VM [CPU snapshot path, memory snapshot path]
(4) RPC server port & pid of VMM process
Backend

Figure 3: Flow for resume VM

6 Deduplication

VM snapshot tend to be large in size. To reduce the storage space required for storing the snapshots, we use deduplication. Deduplication is a process of removing duplicate data from a data set. It is a process of finding duplicate data and storing only one copy of the data. This process is also known as data de-duplication, data deduplication, or content-based deduplication. We start with an observation associated with dropbox. Link to the dropbox paper. blog. Some users experienced a lot less time when uploading files to DropBox. This was because DropBox was able to detect duplicate files and only uploaded the new file. The schema can be seen in figure 4.

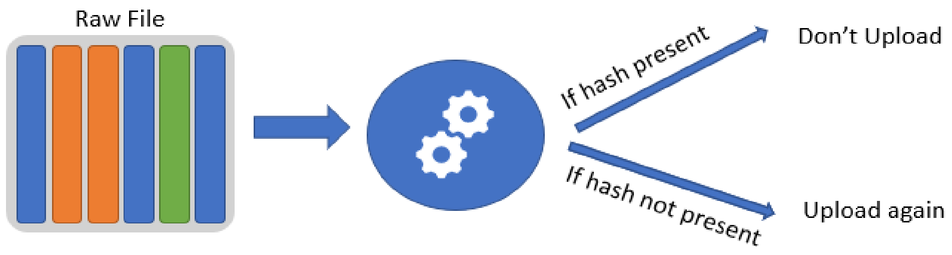

Figure 4: Dropbox deduplication

This can be extended to another design where rather than taking entire file we can break file into chunks and store them in a database. Duplicates need not be stored. Only single copy is sufficient. We design 2 functions. One is to store the chunks and other is to retrieve the chunks.

save_file

This function takes ‘filename‘ as an argument and saves the file into storage. The function breaks the file into chunks and removes duplicate chunks. If a chunk is already present it need not store the copy. The schema can be seen in figure 5.

load_file

This function takes ‘filename‘ as an argument and loads the file from storage. The function collects all the chunks and combines them to form the file.

The schema can be seen in figure 6.

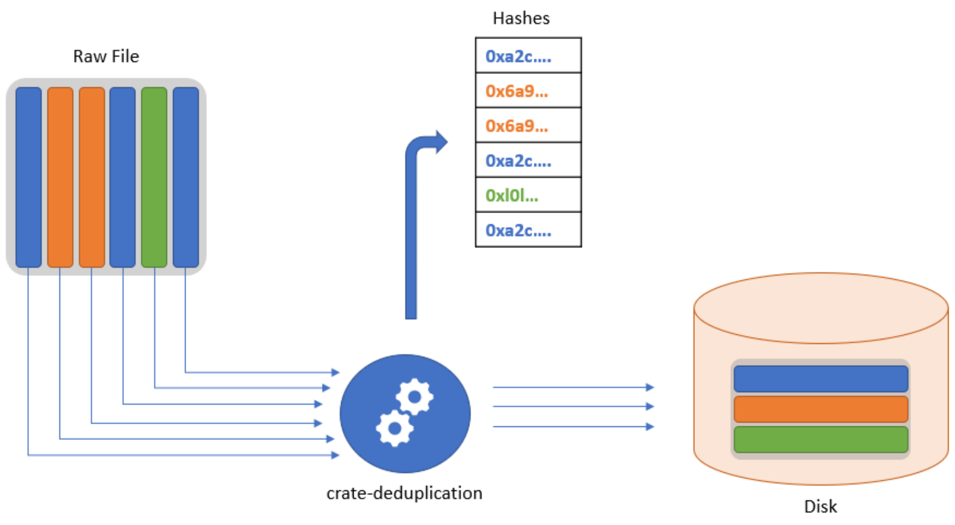

Figure 5: save_file

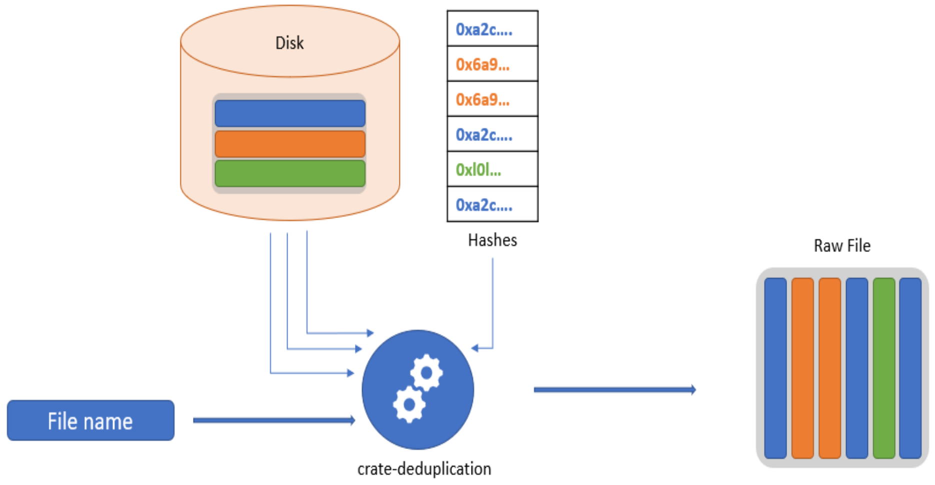

Figure 6: load_file

7 Code

We used version control to manage the code. The main code is available at `https://github.com/anirudhakulkarni/Live-Snapshot`. The code is distributed across various branches as follows:

- **main** : contains the latest stable code
- **benchmark** : contains the code for benchmarking
- **ssh-merge** : contains the code after importing code for ssh team
- **saving-vcpu**: contains the code after implementing saving of vcpu state

Apart from the main repository, the code has following components:

- **deduplication :** `https://github.com/anirudhakulkarni/de-duplication` : The deduplication crate which is responsible for deduplicating the memory and cpu snapshots.
- **webserver :** `https://github.com/chintansheth1711/col732_project_webserver` : The webserver which is responsible for receiving the API requests from the backend and sending the appropriate messages to the VMM.

We also did bunch of experiments to test our hypothesis. The code for the experiments is available at:

- **Live-Snapshot-Simple :** `https://github.com/anirudhakulkarni/Live-Snapshot-Simple` Barebone VM demo with cpu and memory serialization
- **Live-Snapshot-Experiments :** `https://github.com/anirudhakulkarni/Live-Snapshot-Experiments` Modified VMM reference with various breakpoints, additional testing code and modified pipeline

8 Challenges

- Sending interrupt to vCPUs to take snapshots and later resume the VMs from the same state on which they are interrupted. We solved this by sending SIGRTMIN interrupt signal to all the vCPUs, which had made them to exit from their current execution. The signal handler for the same has been setup during the initialization of VM. VM consists of multiple vCPUs, so exits of vCPUs are synchronized via MPSC queue. Snapshot is taken as soon as all vCPUs are exited.

  **vCPU needs to be in the state, in which it is about to exit.
- One VMM one VM. Vmm-reference supports just one VM per VMM process, and for creating multiple VM's we need to spawn multiple VMM processes. Rest API has been exposed for creating VM's via web server.
- Drainage of CPUs. Vmm-reference drains out all CPUs after exiting from run loop. We use Mutexes to transfer ownership
- Lack of serialization. Vmm-reference does not supports serialized structs. We serialize each struct.

9 Results

We were successfully able to boot bzimage of general linux (focal fossa), take snapshot of it and restore vm state from saved snapshot. On i79750-H processor with 8GB RAM here are the benchmarks for a VM having 2 CPUs and 256MB RAM:-

	Average over 100 runs	Standard deviation over 100 runs
Boot time	881.55 ms	44.507 ms
Restore from snapshot	300.26 ms	57.046 ms
Snapshot time	2.82 sec	0.56sec

Upon increasing memory size, time taken to take snapshot and restore time both increases linearly. Here is the plot showing the increment of save and restore time with respect to memory size.

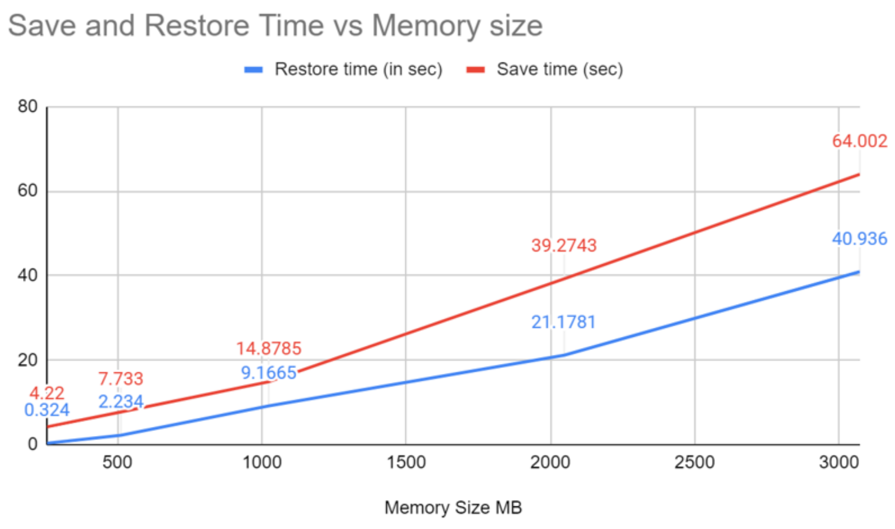

Figure 7: Save and Restore time vs Memory size

Deduplication results:

1. We were able to successfully store the snapshot in a efficient way
2. For saving 150 snapshots of 256MB each, we were only taking 2.4 GB disk space
3. Upon increasing chunk size, the reduction in size by deduplication decreases. Here's a plot showing this behaviour:-

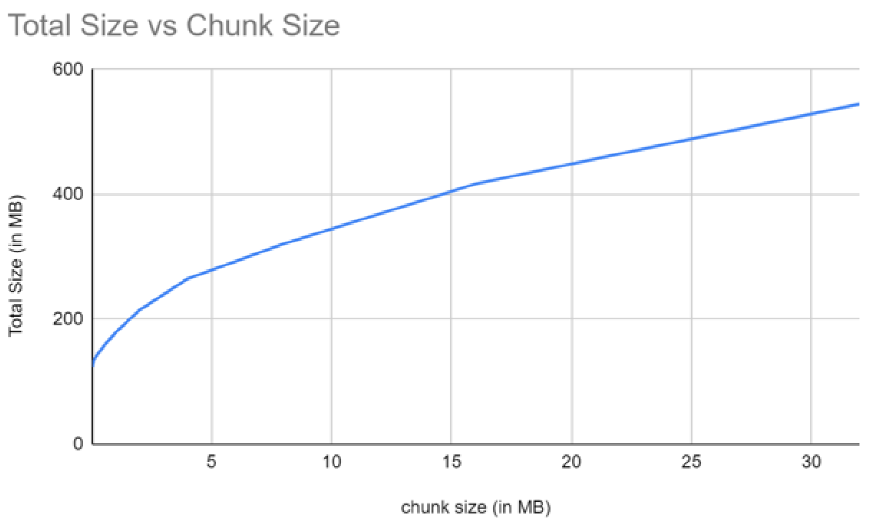

Figure 8: Total size vs Chunk size

The graph was plotted for saving 4 snapshots of 256MB each. It is clear from the graph that upon increasing chunk size, size taken by snapshot increases. This is because, larger chunk size would imply low number of duplicate chunks

4. From the graph it can also be inferred that, for saving 4 sanpshots of 256MB each we were able to save them in as low as 90 MB in 1.3s giving 91% reduction

5. As the chunk size decreased, number of chunks increased. This increased the time taken to save and load the file. The results are as follows:-

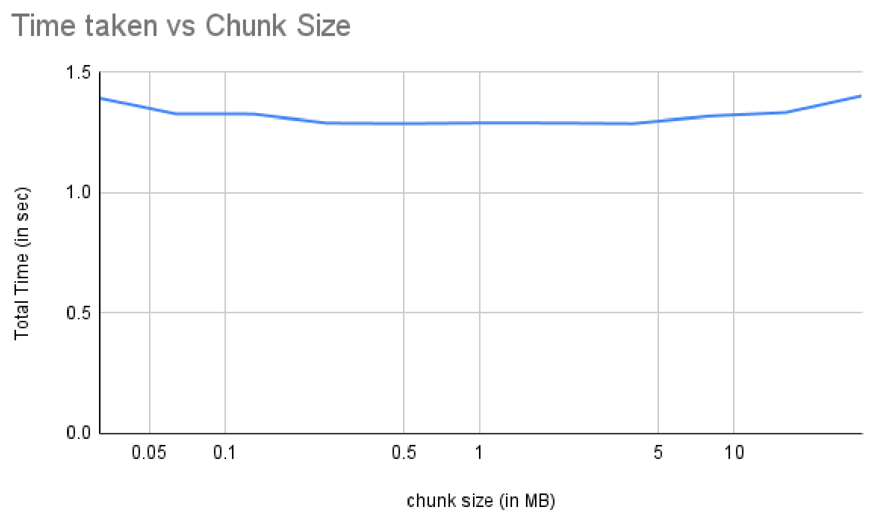

Figure 9: Time taken to save a file

6. Extreme examples of chunk size trade-off are 1 byte and 1GB. The later is inefficient as the library is not able to de-duplicate the files at all. The former is inefficient as the library is not able to de-duplicate the files efficiently.

10 Limitations

VMM and web server needs to be on the same server. Since the web server is creating the VMM process by using fork-exec on receiving requests for creating a new VM, it needs to be on the same server on which it can create new VM's.

11 Future Work

- Looking at the effectiveness of the deduplication library, we ~~plan to~~ release library as rust crate for general purpose use. Find it on `https://crates.io/crates/deduplication` Now you can install it via

```
cargo install deduplication
```

- The project can be exteneded to support multiple VMs per VMM process. This would require a lot of changes in the VMM reference code.
- The project can be extended to support additional devices and arm architecture

12 References

- Rust Documentation: `https://doc.rust-lang.org/book/`
- Firecracker Documentation: `https://github.com/firecracker-microvm/firecracker/blob/master/docs/`
- Articles on DropBox Malpractice:
  - `https://blog.fosketts.net/2011/07/11/dropbox-data-format-deduplication/`
  - `https://news.ycombinator.com/item?id=2478595`
- Vmm-reference Documentation: `https://github.com/rust-vmm/vmm-reference/`