Reducing Data Transfer Overheads during Live Migration of Application Containers

Yashas D[1] and Sawan H N[1]

IIT Kanpur[1], Debadatta Mishra[2], Shiv Bhushan[3]
yashasd21, sawanhn21@iitk.ac.in

Abstract. Application containers are commonly used for application deployment in contemporary cloud computing environments. Container migration across different hosts enables cost-effective cloud management by supporting better server consolidation, system maintenance without elongated service outage, efficient load balancing etc. In iterative pre-copy migration, a widely used migration technique, only the modified memory pages are transferred over multiple iterations while the container remains alive till the final iteration. In this method , the amount of data transfer after every iteration depends on the number of modified pages as the current OS support for memory modification tracking is at a page granularity. In this project, we empirically demonstrate that the data transfer size can be significantly reduced if we reduce the granularity for pages that are not changed significantly. Further, we show that if the exact changes for the pages that have minor modifications are transferred, the amount of data transfer can be reduced for real-life memory intensive workloads such as Redis.

Keywords: Operating system · CRIU · Migration

1 Introduction

Cloud computing has revolutionized the way businesses deploy and manage applications by offering scalable resources and flexible service models. As organizations increasingly adopt cloud-based infrastructures, the ability to efficiently move running applications and services between different cloud environments becomes paramount. This capability, known as live container migration, is essential for optimizing resource utilization, balancing loads, managing maintenance without downtime, and implementing robust disaster recovery strategies. One of the key technologies facilitating live container migration is Checkpoint/Restore in Userspace (CRIU).

CRIU is an innovative open-source tool that enables the freezing of a running container and the capture of its entire state, including memory, processes, and network connections, into a persistent snapshot. This snapshot can then be transferred and restored on any other system that supports CRIU, effectively resuming the container's operation without losing its state. The integration of CRIU into cloud computing environments allows for seamless transitions between

different platforms and clouds, enhancing the flexibility and agility of application management. By utilizing CRIU for container migration, businesses can achieve higher availability, better fault tolerance, and continuous service delivery across their cloud infrastructures.

Pre-copy is a migration technique used primarily in virtual machine (VM) and container migration to minimize downtime and enhance performance. It operates by first copying all the memory contents of a VM to the target host while the VM is still running on the source host. Subsequent rounds copy only the memory pages that have changed since the last copy. This iterative approach reduces the total amount of data transferred and the downtime required for the final switch-over, making it a preferred method for live migrations in dynamic computing environments.

CRIU presently checkpoints by sending over data at page granularity(4KB). Modification of a single bit of a page will lead to the entire page being sent which is a huge memory overhead. Since the downtime of migration directly depends on the amount of data that has to be sent reducing the amount of data sent can significantly decrease the downtime

We introduce an approach which the modified pages could be either sent at page granularity or at a lower subpage granularity. The pages that have been modified greatly are sent entirely while the pages that have been slightly modified are sent through a structure called bitmap where only the modified bits are sent at the defined subpage granularity.This ensures that only the modified parts and not the entire page is sent saving a lot of memory.To check the number of bits that have changed from one iteration to other we maintain a buffer that stores the pages that are most likely to be modified in the next iteration and at the end of every iteration we compare the updated data with the data in the buffer if its present and then decide whether to send the entire page or send the modified subpages. Deciding the pages that are to be buffered is a major challenge since we have to predict them before the start of every iteration.

Keeping in the mind that the size of the buffer has to be small so as to not give additional memory overheads and having predicted the pages that are to be kept in the buffer we are ready to go.On benchmarking with a real time work load YCSB on redis container significant reduction in data transfer was observed.

2 Background and Motivation

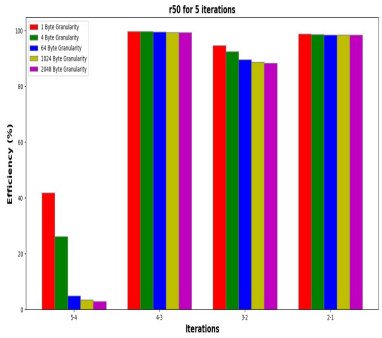
Pre-copy iterative migration, as implemented by CRIU (Checkpoint/Restore in Userspace), is a sophisticated approach to reducing downtime in live migrations of containers or virtual machines. Initially, CRIU captures and transfers a full snapshot of the memory state of the process or container. In subsequent rounds, only the memory pages that have been modified since the last snapshot are transferred. This incremental approach minimizes the data needed to be sent over the network, reducing bandwidth usage and migration time. Recent enhancements in CRIU's pre-copy mechanism focus on optimizing these iterative transfers by

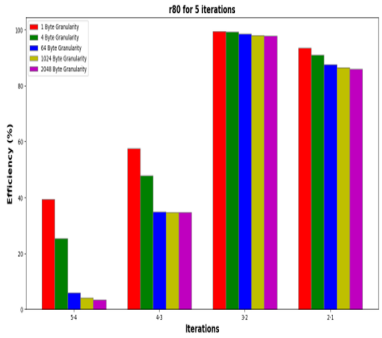
improving the algorithm that detects changed memory pages, thereby reducing the transfer volume even further. Additionally, advancements have been made to minimize the freeze time during the final state transfer, ensuring that the process or container resumes quickly on the target system with minimal disruption.

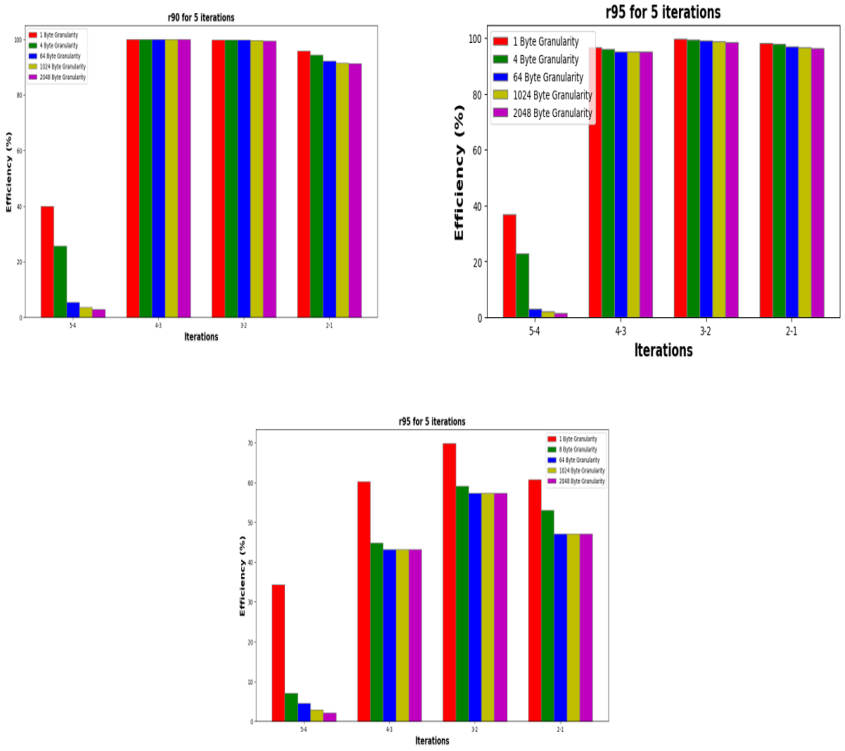
Reducing data transfer between iterations in pre-copy iterative migration is crucial for several reasons. Firstly, it minimizes network bandwidth usage, which is especially important in environments with limited network resources or high network costs. Secondly, it enhances the efficiency of the migration process, allowing systems to handle more migrations simultaneously with reduced impact on performance. This is particularly valuable in cloud computing and data center environments where frequent migrations are common. Lastly, reducing data transfer helps in achieving quicker migration completion times, thereby minimizing downtime and improving the availability and resilience of services. This is vital for maintaining service continuity and user experience in high-availability systems.

3 Motivation

The main motivation was the fact that there will certainly be pages which will not be modified much after every iteration and keeping them in the buffer would save a lot of data transfer.
Specifications:YCSB workload on redis container with zipfian distribution and data size 1 KB records (10 fields, 100 bytes each, plus key).It was run on a virtual machine with Ubuntu 22.04.3 LTS.
The first 4 graphs are plotted for different read write percentages on the active process that was doing the read write operations while the last graph is for the redis server itself.Since server does more write operations too we don't see the reductions to be as high as in the case of the active process.On the y axis is plotted the percentage of data we could save if data transfer occurs at those sub page granularities. Hence even if we cannot tap all these pages in the buffer we could still get significant efficiency if we manage to predict most of the pages correct.







Ensuring optimum buffer efficiency would help us achieve the above shown data reductions but since our buffer size is limited we cannot store all the pages but have managed to reach acceptable efficiency.

4 Design

Though the design and approach sounds optimal there are certain challenges our approach faces.

4.1 Challenges

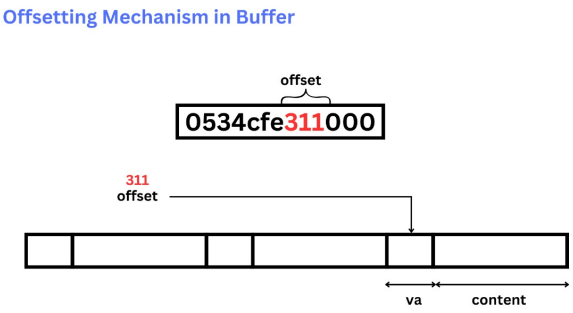
One of the major challenge is to decide the page replacement policy in the buffer. Since different processes behave differently it is difficult to implement the same policy for every process and also since the process behaves differently across iterations there has to be different policies for different iterations too to achieve maximum efficiency. It was observed that pages are getting modified uniformly across the entire address space and some sort learning mechanism that learns from previous iterations would require us to store certain meta data for every page as in the number of times it has been dirtied before or the number of pages in its close proximity that have been dirtied.

The second challenge is how it would eventually affect the downtime of migration. Since after every iteration we are running our page replacement policy and comparing the contents of pages individually it would take some time.But the amount of reduction we would achieve due to reduction in data transfer might overweigh this. This has to ben tested thoroughly to decide other parameters to ensure optimal behavior.

Also the nature of the work load if it is read intensive or write intensive and how frequently it is updating its data is also a reason of concern that has to be monitored carefully.

4.2 Design approach

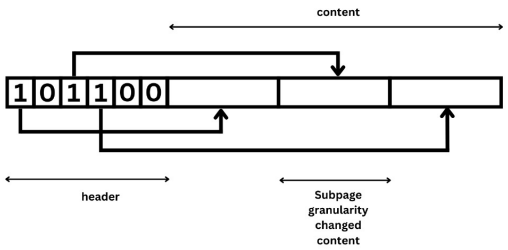
Keeping the above challenges in mind we have come up with a basic design that doesnt store meta data as of now and a simple page replacement policy.We have used two additional structures a buffer to cache pages and bitmap to send pages at subpage granularity.



The entire buffer is divided into blocks of (6+4096)bytes.The first 6 bytes of a block store the virtual address of the page and the subsequent 4096 bytes hold the content of the page.The offsetting mechanism is as follows. We calculate the number of bits required to offset the buffer depending on the buffer size. We then extract those many bits from the virtual address of a page leaving out the 12 lsb bits which are always 0 as addresses are at page granularity and choosing the next required number of bits. So multiple pages can have the same block in the buffer if they have same offsetting bits.

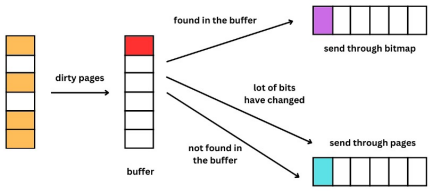
The structure of a bitmap consists of two parts,the header and the content.The header is of size page size divided by the chosen subpage granularity. Every subpage is allocated a bit that states if the subpage is modified and hence present in the bitmap or not. The content part consists of all the modified subpages in order.

Structure of Bitmap



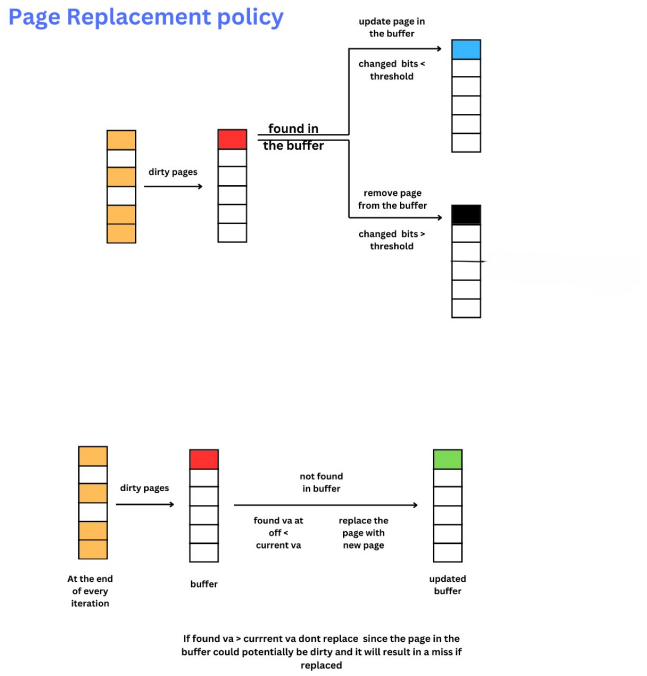
After the first iteration since we do not have anything in the buffer to compare we just populate the buffer by traversing all the dirty pages.In the subsequent iterations after every iteration we compare the data and decide if to send it through bitmap and also decide to keep it the buffer or replace a page thats already there. Given below is the criteria used to decide if to send it through bitmap or not.

Deciding which method to send the page



Given below is the page replacement policy where threshold is basically page size - len of the header of bitmap so as to ensure we are not sending more bits than a page.

There is no point in keeping the page that has significantly changed since it is probable that it shows the same behaviour again and there is no use keeping it in the buffer. So we remove it to give space for new page and not removing if va already in buffer is greater than the current va is because it is possible that the one in buffer is also dirty and if we replace it it might lead to buffer miss so we avoid that.



4.3 Implementation details

The following is a very brief CRIU code snippet which both updates the buffer and decides if to a send a page through bitmap or not

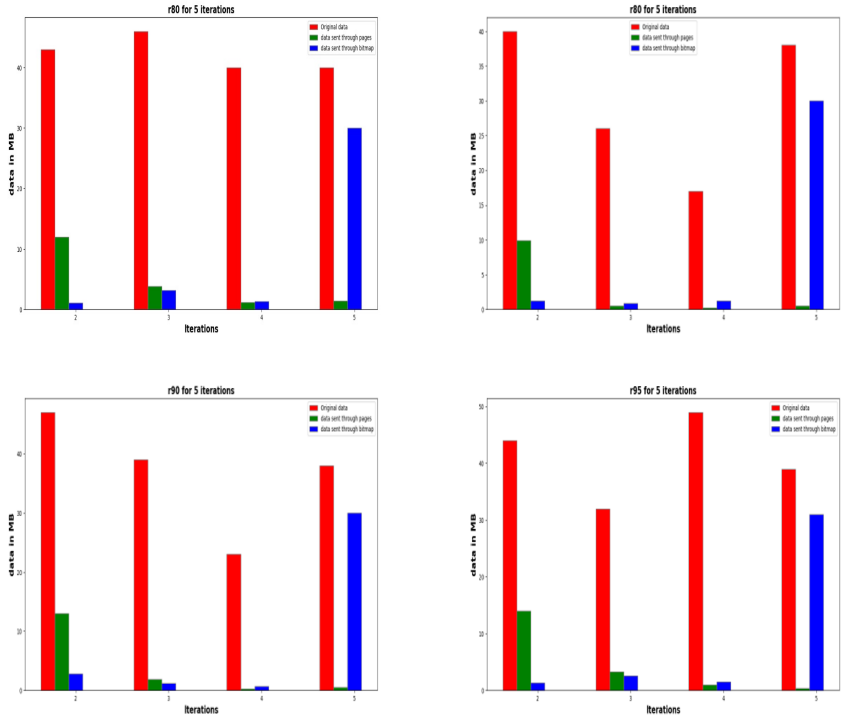
```
while(nr != 0){
    // process each page separately to check if we have it in buffer
    unsigned char *buff = (unsigned char *)xmalloc(PAGE_SIZE);
    ret = read(pipe_fd,buff,PAGE_SIZE);
    found_ptr = hash(va , lsb_bits , block_size , global_ds);
    found_va = read48BitNumberFromMemory(found_ptr);
    if(found_va == va){ // checking if its already present in the buffer
        // writing for those continuous pages not present in the buffer
        bits_changed = compare_memory(found_ptr + 6, buff, PAGE_SIZE , 4);
        total_bits_changed += bits_changed;
        if(bits_changed > THRESHOLD){ // A lot of bits have changed
```

```
            // send the entire pages in pages.img
            written_bits = write(img_raw_fd(xfer->pi), buff, PAGE_SIZE);
            // hit case-remove the page
            write48BitNumberToMemory(found_ptr, 0);
            if(nr_cont_pages == 0){
                cont_va = va;
            }
            nr_cont_pages++;
        }
        else{
            // send the diff in bitmap.img
            if(nr_cont_pages > 0){
                iov_dup.iov_len = nr_cont_pages*PAGE_SIZE;
                iov_dup.iov_base = decode_pointer(cont_va);
                ret = xfer->write_pagemap_sgt(xfer, &iov_dup, flags, 1);
                nr_cont_pages = 0;
            }
            // hit case-update the page
            ret = update_memory(found_ptr + 6, buff, PAGE_SIZE);
            ret = xfer->write_pagemap_sgt(xfer, iov, flags, 0);
            if(ret != 0)
                return -1;
            ret = xfer->write_bitmap_sgt(xfer, found_ptr+6, buff, PAGE_SIZE);
        }
    }
    else{ // page is not present in the buffer
        // send the entire page in pages.img
        non_buffered_pages++;
        written_bits = write(img_raw_fd(xfer->pi), buff , PAGE_SIZE);
        if(nr_cont_pages == 0){
            cont_va = va;
        }
        nr_cont_pages++;
        // page replacement policy
        if(va > found_va){
            // update the buffer with the new page
            write48BitNumberToMemory(found_ptr, va);
            ret = update_memory(found_ptr + 6, buff, PAGE_SIZE);
        }
    }
    nr--;
    va += PAGE_SIZE;
    xfree(buff);
```

```
55 }
```

Listing 1.1. C code snippet

5 Results



The above graphs were obtained for different read write percentages and it shows that our approach greatly reduces the data that has to be sent after every iteration.We have tuned the subpage granularity to be 4 bytes looking at many parameters like last iteration data reduction and average data reduction across iterations.8 bytes was giving the closest efficiency that was slightly lesser than 4 bytes approximately 5 percent lesser efficiency. In read intensive work loads our algorithm is highly efficient since there is very little write happening and most of the pages marked dirty can be sent through bitmap. The buffer size is 64MB and it has an average space efficiency of 98 percent.The conflict misses at every offset of the buffer was mostly uniform indicating pages are being modified randomly. Even is write intensive YCSB workload our approach is performing quite good considering the amount of reduction observed.Most of the pages that are getting modified are present in the MAP_ANONYMOUS region. The significant reduction in data volume is because a lot of pages that are marked PE_PRESENT

have very few bits changed and that might be because of the change of permissions of an entire area of mmapped pages and in the earlier approach we still sent the entire pages but now since we send only the changed parts we send very little. In the last iteration just before dumping not a lot of reduction has occurred because of the rigorous changes just before final dump but still slight reduction in the last iteration is also favourable. Storing additional metadata for each page would be a huge memory overhead but it can be done to improve buffer hits and hence lesser data transfer.

6 Conclusion

In this approach we have introduced a mechanism where dirtied pages could be sent at subpage granularity using the buffer structure. Page replacement policy could be worked upon since ours is a basic policy and including kernel support to actively send the pages when they are dirtied for the first time would improve buffer efficiency.These would be done in the next part of the project.

7 Credits

We would like to thank instructor incharge Debadatta Mishra for his support throughout the project in providing valuable insights and our mentor Shiv Bhushan for his immense support throughtout in understanding the basics of CRIU and understanding how it works and everything in general.