Developing a precise and efficient Algorithm for Adaptive Checkpoint Scheduling of Virtual Machine Replication on Multi-core Systems

Final Report, Final Year Project 2017-18

Updated on 15 April 2018

Version 1.3

Source Code : https://github.com/angad2102/Plover/tree/Adaptive-Plover

Angad Singh (3035124764)

Project Advisor: Dr. Heming Cui

Abstract

High availability, which is the ability to recover quickly after a hardware failure, is hard to achieve yet desirable. Nowadays, high availability is achieved through virtual machines using a checkpoint mechanism where changes are propagated to a backup machine at a fixed checkpoint (up to 40 times in a second). This can produce unnecessary load on the system especially multi core systems. Previous work has shown that adaptive checkpointing is a more efficient solution when modeled around the properties of the software being run on it. The objective of this project was to produce an algorithm for adaptive checkpointing which can be integrated into existing solutions such as Plovcr. The project provides a comprehensive workload analysis on Plover with variable checkpoint epoch lengths. The algorithm developed adapts to the workload on the virtual machine. The algorithm has shown promising primary results in improving the run time.

Acknowledgment

I would like to extend my gratitude to my FYP advisor, Dr Heming Cui, for helping me with the planning and execution of the project, Cheng Wang for providing me all the support and resources I needed and Cezar Cazan for providing valuable feedback on my work and helping me improve the quality of my work.

Table of Contents

List of Tables

List of Figures

Abbreviations

VM	Virtual Machine
CPU	Central Processing Unit
PC	Personal Computer
I/O	Input/Output
GB	GigaByte
SSD	Solid State Drive
GHZ	Gigahertz
TB	Terabyte
GBPS	GigaBytes per second
TCP	Transmission Control Protocol
IP	Internet Protocol
RDMA	Remote Directory Memory Access
FTP	File Transfer Protocol
PHP	Hypertext Preprocessor

I. Introduction

Recently, there has been an increasing demand for online service deployment on virtualized infrastructure [1]. Online services are processing more and more requests concurrently nowadays which require virtual machines (VM) to utilize more and more virtual CPUs on multi-core hardware. Because of this rise in cloud computing, hardware failures have become more common [5]. The need for high availability is rising.

However, high availability is hard to obtain. Previously, high availability was only possible using commercial hardware or application specific replication software [2]. Several Solutions have been given in the past to make high availability common place. One of the most common solutions is checkpoint recovery. The entire state of the Virtual machine is copied at very high frequencies and the changes are propagated to the backup virtual machine almost instantaneously. While the changes are being copied, the virtual machine runs speculatively and no output is being released to the user. The output is released when the two machines have been synchronized successfully. This process is carried out at fixed intervals [5].

The problem with this method is that it can cause significant overhead because of the large amount of data that needs to be copied and transferred, even on uniprocessor VM setups, so frequently [3]. Research (eg [3], [4]) has suggested that adaptive checkpointing instead of fixed frequency checkpointing can improve the system performance.

The main aim of this project is to produce an algorithm for adaptive checkpointing to decrease the overhead, especially on multi-core systems and effectively improve the system performance. The algorithm has been has also been implemented using PLOVER [1]. The source code is available at https://github.com/angad2102/Plover/tree/Adaptive-Plover.

The rest of this report is organized as follows. Section II covers the background of the project. Section III covers Literature Review of four papers related to this field. Section IV describes the detailed methodology of the project. Section V describes the algorithm. Section VI talks about the evaluation of the algorithm and results. Section VII discusses the future plans for the project and section VIII concludes this report.

II. Background

High Availability is defined as the ability of a system to be operational and fully functional for a suitable length of time [8]. To achieve High Availability we need to make sure that there is a backup for the system which can be accessed at any time without any major delay in case of a hardware failure and the web service can be resumed from the point where the system crashed. There should ideally be no data or network packet loss. This can be achieved by using commercial hardware built for this purpose or restructuring the code of each and every application in the virtual machine to include complicated logic for recovery [2]. However, these solutions are expensive and are not available to the masses. Another approach would be to propagate the system state synchronously to a backup machine, but this will slow down the system and the systems memory throughput would be comparable to a replication performing network device, which is not desirable.

Remus [2] provided a very good solution to this problem by replicating the system state at frequent predefined checkpoints. It lets the host system perform its computations in speculative state and propagates the changes asynchronously to the backup machine at every 25 ms [2].

COLO (Coarse Grain Lock Stepping) is another solution to keep the primary and backup systems in sync [14]. It compares the generated output from both the systems and only executes a checkpoint if the output differs. However analysis has shown that COLO's performs severely deteriorates when there are multiple client connections [1].

Plover is a new system which implements adaptive checkpointing depending on the load on the system. Incoming output is supplied to both the primary machine and the secondary machine. A checkpoint is issued when the primary system detects that the secondary system is in an idle state which increases efficiency and reduces idle system time [1]. However experiments carried out in this project showed that the system can carry out very frequent and unnecessary checkpointing causing excessive overhead.

III. Literature Review

i. "Remus: High Availability via Asynchronous Virtual Machine Replication" [2]

Remus [2] adopts a frequent checkpoint backup model to maintain High Availability. Remus keeps a backup host and propagates the changes to this backup host from the primary at a predefined interval.

The network output from Remus is stored in the buffer until the system state synchronizes with the backup's state. The checkpoint is very frequent, at every 25 milliseconds [2] (Which implies a rate of upto 40 checkpoints in a second). Disk changes are propagated to the backup asynchronously since the entire disk snapshot needs to be transferred. The output is only released once the two machines have synchronized and a confirmation from the backup machine has been received. This essentially takes care of the output commit problem [2] which means that any system state which has already been displayed should be recoverable [6]. When a failover occurs, the backup is started and replaces the primary. However, it is important to note that the virtual machine does not do any execution until a failover occurs [2].

Due to the amount of data which needs to be transferred this task can prove to create a significant overhead even on uniprocessors [3]. On multi-core systems the overhead is even more because of more number of cores working concurrently, hence increasing the workload per time unit.

ii. "PLOVER: Fast and Scalable Virtual Machine Fault Tolerance on Multi-core" [1]

State machine replication (SMR) enforces the same total order of inputs for a service which is replicated along hosts. By doing this most of the memory pages in the two hosts get updated and thus do not require to be transferred from the primary to the secondary host.

This paper discusses the system PLOVER which is the first Virtualized State machine Replication (VSMR) System. VSMR enforces the same total order of inputs across replicated virtual machines [1]. This keeps majority of the memory pages updated and equivalent, and very few divergent pages have to be transferred across systems.

Plover implements an adaptive checkpointing model in which a checkpoint is carried out by the primary system only when it detects the secondary system to be in an idle state. However when the system is not under load or is idle, the frequency of the checkpoints can be very excessive.

iii. "Workload Adaptive Checkpoint Scheduling of Virtual Machine Replication" [3]

This paper discusses adaptive checkpointing based on an analysis of workloads of several applications. This paper attempts to reduce the overhead by dynamically adjusting the checkpoint frequency based on properties such as the number of dirtied memory pages, the number of disk I/O operations, the number of transferred network packets and the network bandwidth available for replication [3].

Apart from the adaptive checkpoint scheduling, the paper also implements a fine grained copy-on-write mechanism to avoid the downtime caused by checkpointing [3]. This has been achieved by only locking the memory pages of which the values have to be transferred to the backup machine, allowing the virtual machine to run concurrently [3].

iv. "Adaptive Remus: adaptive checkpointing for Xen-based virtual machine replication" [4]

This paper attempts to decrease the overhead by adapting the checkpoint frequency according to the application being run on it. It suggests the virtual machine to run in two modes. (i) Network mode: where it increases the checkpoint frequency where high output network traffic is detected. (ii) Processing mode: where it decreases the checkpoint frequency because of the low output network traffic [4].

IV. Methodology

The project was carried out primarily in two phases. First phase consisted of workload analysis of several different applications on PLOVER. The applications selected for this analysis were Mongoose, Apache (using PHP), Tomcat, FTP (File Transfer Protocol) server, and NAS parallel benchmarks. The applications were tested with varying intervals for checkpoints ranging from 10ms to 500ms. Properties such as dirtied memory pages and runtime were compared. After this analysis, an algorithm was developed to reduce the system overhead and runtime. Phase two consisted of integrating the algorithm with PLOVER and comparing its performance with PLOVER.

The reason for choosing PLOVER was that it is one of the most recent solutions for high availability and its runtime is considerably better than REMUS or COLO and is more scalable than the 2 systems [1].

All the tests and experiments have been done on Dell R430 servers with Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. All NICs are Mellanox ConnectX-3 Pro 40Gbps connected with infiniband [10]. The ping latency between every two replicas is 84 microseconds (the TCP/IP over RDMA round-trip latency).

The rest of the section talks about the workload analysis conducted.

i) Workload Analysis

The workload analysis was performed on PLOVER with varying checkpoint frequency from 10ms to 500ms. Mongoose, Apache (using PHP), Tomcat, FTP (File Transfer Protocol) server, and NAS parallel benchmarks were run on the virtual machine. The number of memory pages dirtied and runtime were collected and compared.

a) Mongoose

Apache's HTTP server benchmark tool, ab, was used to send 128 requests with a concurrency of 16 requests being sent at the same time to the mongoose server running on the virtual machine. The mongoose server was serving a simple web page with a very small calculation being done on the backend side.

Figure 1.a shows the number of dirtied memory pages and the number of pages which need to be transferred on average at each checkpoint with ranging epoch lengths for checkpoints. The test was carried out 50 times with a different checkpoint frequency interval ranging from 10ms to 500ms (with steps of 10ms).

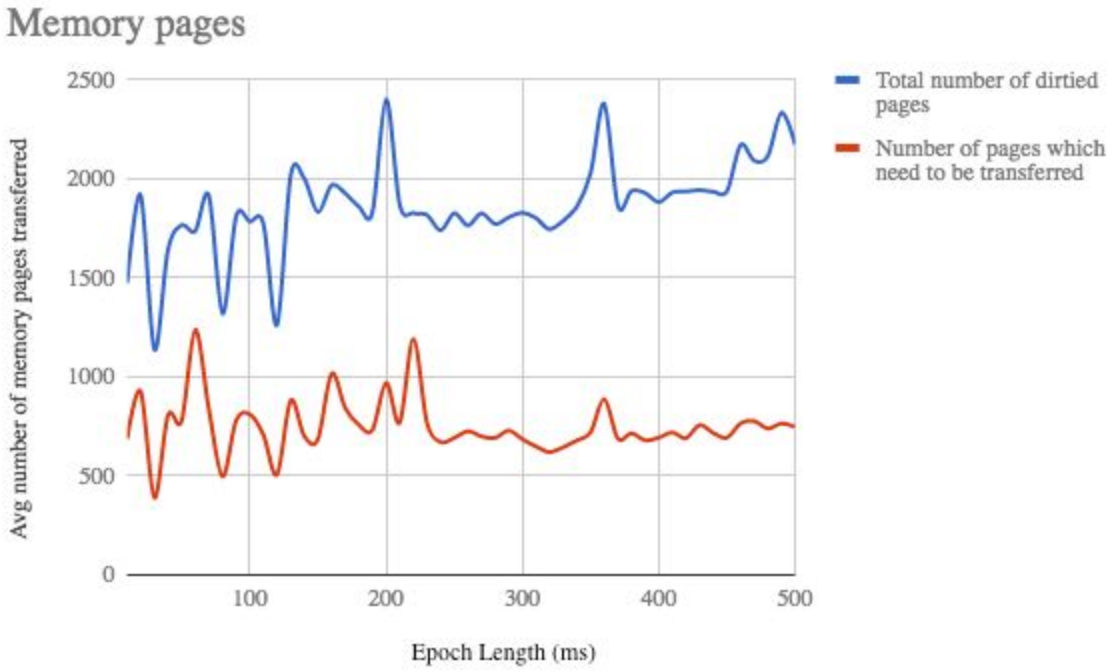

Figure 1.a Mongoose - Dirtied memory pages vs epoch length

The average number of memory pages dirtied and the number of memory pages which need to be transferred at each checkpoint are roughly about the same for all epoch lengths. No Major pattern can be observed from this graph.

Figure 1.b shows the runtime of the experiment in respect to the epoch length of the checkpoints.

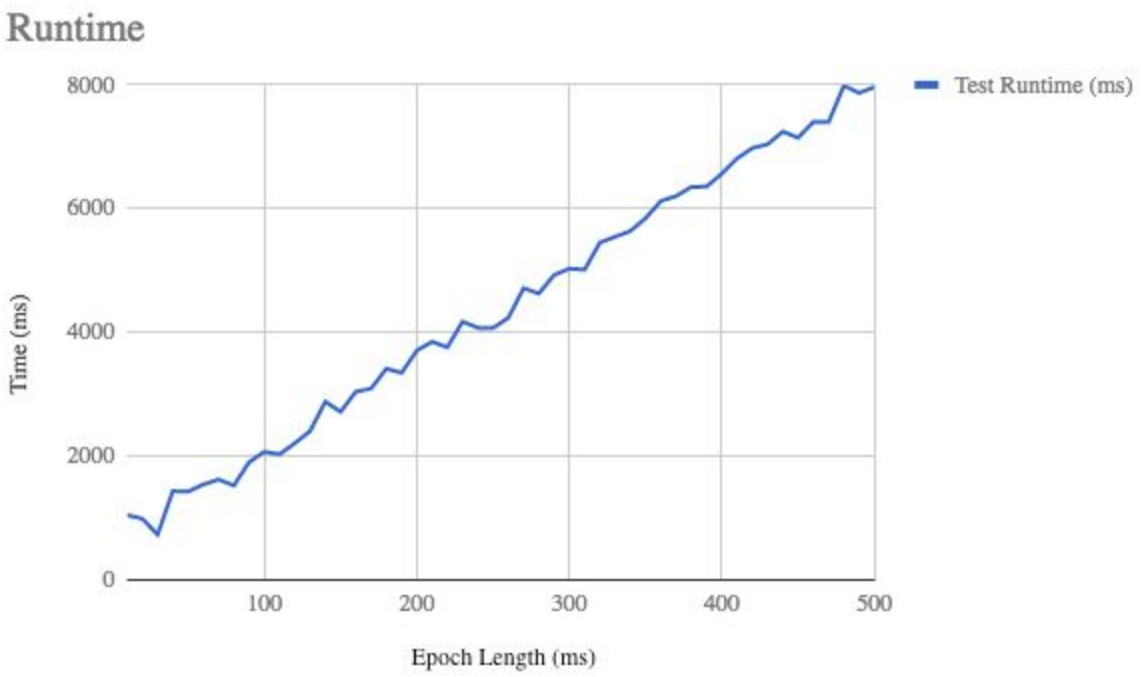

Figure 1.b Mongoose - Runtime vs epoch length

The runtime increases with increasing epoch length. This makes sense as the output is not released till a checkpoint is carried out and our experiment relied heavily on network packet exchange as 128 requests were being sent out.

This indicates that an application which needs to send out network packets quite frequently thrive better at low epoch lengths as compared to higher ones. This produces a load on the replication software as it has to transfer quite a lot more pages cumulatively as compared to if the checkpointing was being done at longer epoch lengths. Though network output is released as soon as possible resulting in reduction of runtime.

b) Apache (using PHP)

Apache's ab tool was used to send 128 requests with a concurrency of 8 requests being sent at the same time to the apache server running on the virtual machine. The php page on the server was doing a time consuming calculation before sending the web page with the response back to the client.

Figure 2.a shows the number of dirtied memory pages in respect to the variable epoch length from 10 ms to 500 ms (with steps of 10ms).

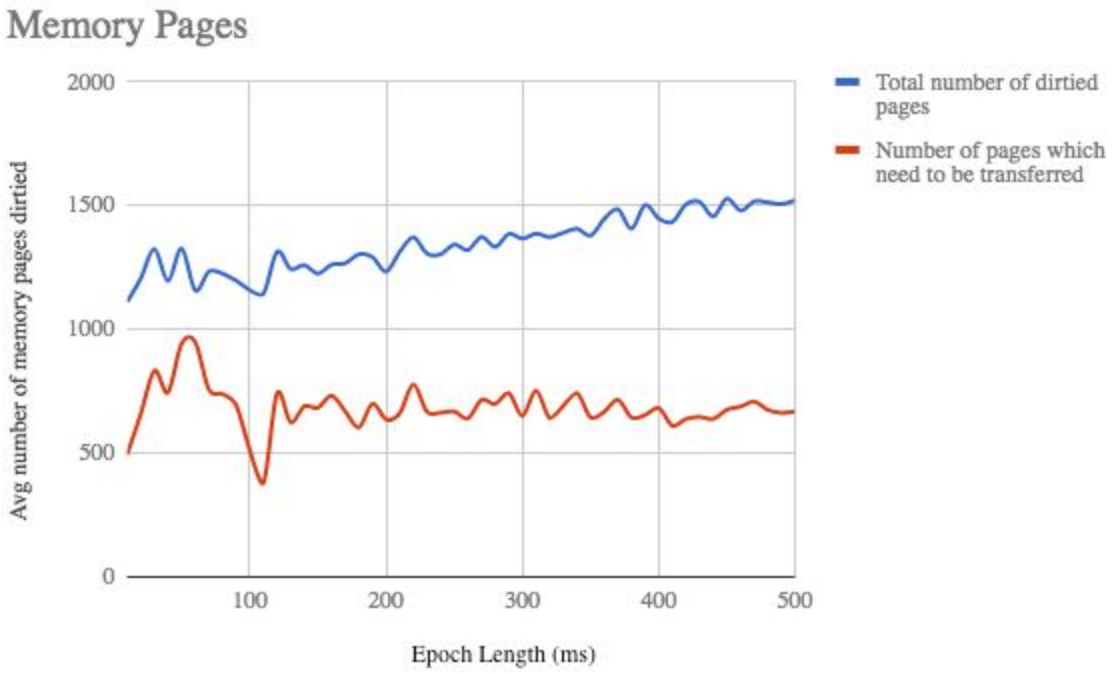

Figure 2.a Apache - Dirtied memory pages vs epoch length

Figure 2.b shows the runtime of the 50 experiments with respect to epoch length.

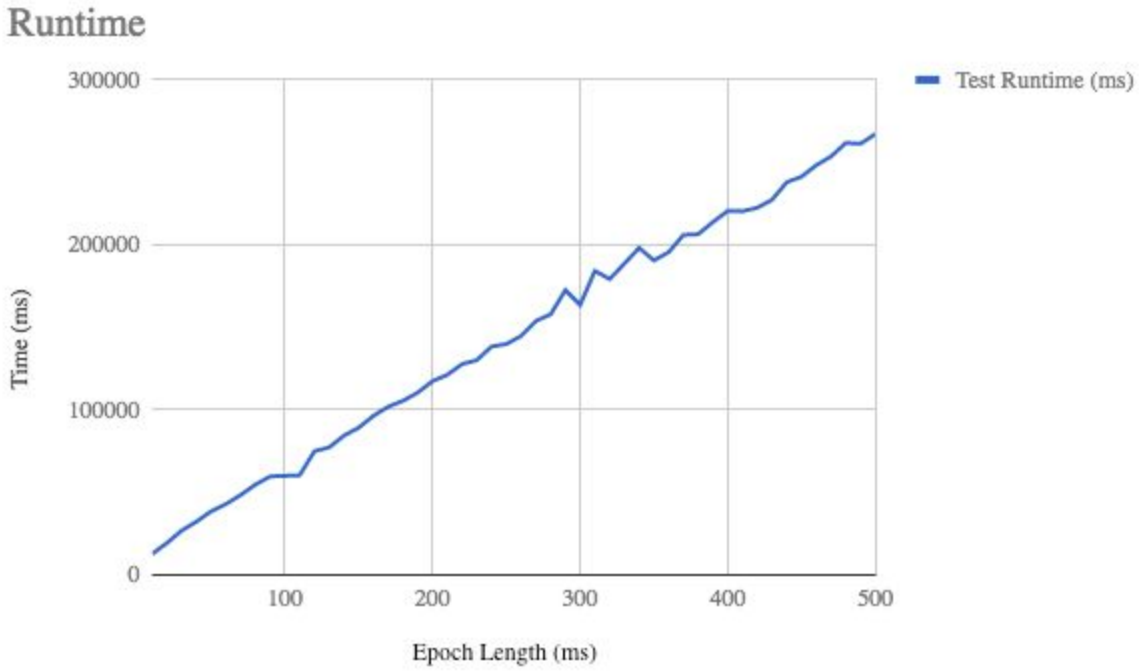

Figure 2.b Apache - Runtime vs epoch length

The experiment showed similar results and similar graphs as the experiments with Mongoose. However in this experiment, a long calculation is a also being done each time and there is both network packet exchange and processor load. In a situation like this an algorithm which switches quickly between high checkpoint frequency and low checkpoint frequency would be ideal.

c) Tomcat

For tomcat, 8 requests were sent by ab all at the same time to the Tomcat server running on the virtual machine. The tomcat server does a small calculation and sends the response back to the client.

Figure 3.a shows the number of dirtied memory pages vs epoch length. The tests were again done with a variable epoch length from 10 ms to 500 ms (steps of 10 ms).

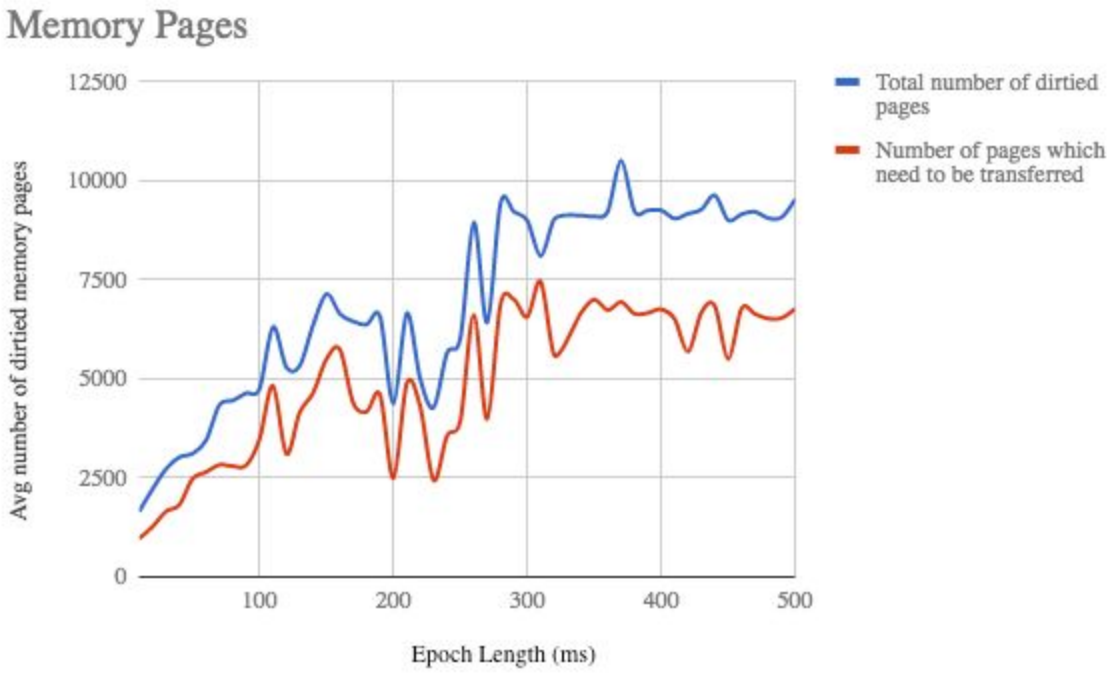

Figure 3.a Tomcat - Dirtied memory pages vs epoch length

The number of memory pages seem to be increasing with increase in epoch length. Figure 3.b shows the runtime of the experiment with respect to the epoch length.

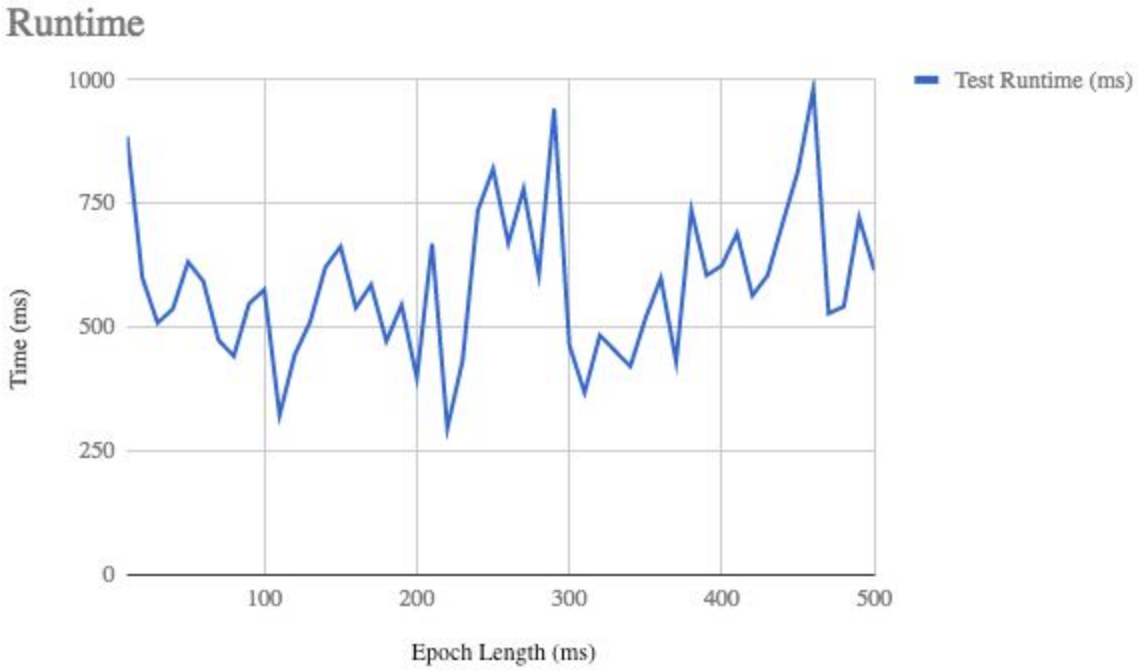

Figure 3.b Tomcat - Runtime vs epoch length

d) FTP (File Transfer Protocol) server

A file of 50 MB was sent to the virtual machine via the FTP server running on it. The experiment was conducted 50 times with varying epoch length from 10 ms to 500 ms (steps of 10 ms).

Figure 4.a shows the number of dirtied memory pages in respect to the epoch length and figure 4.b depicts the runtime with respect to varying epoch length.

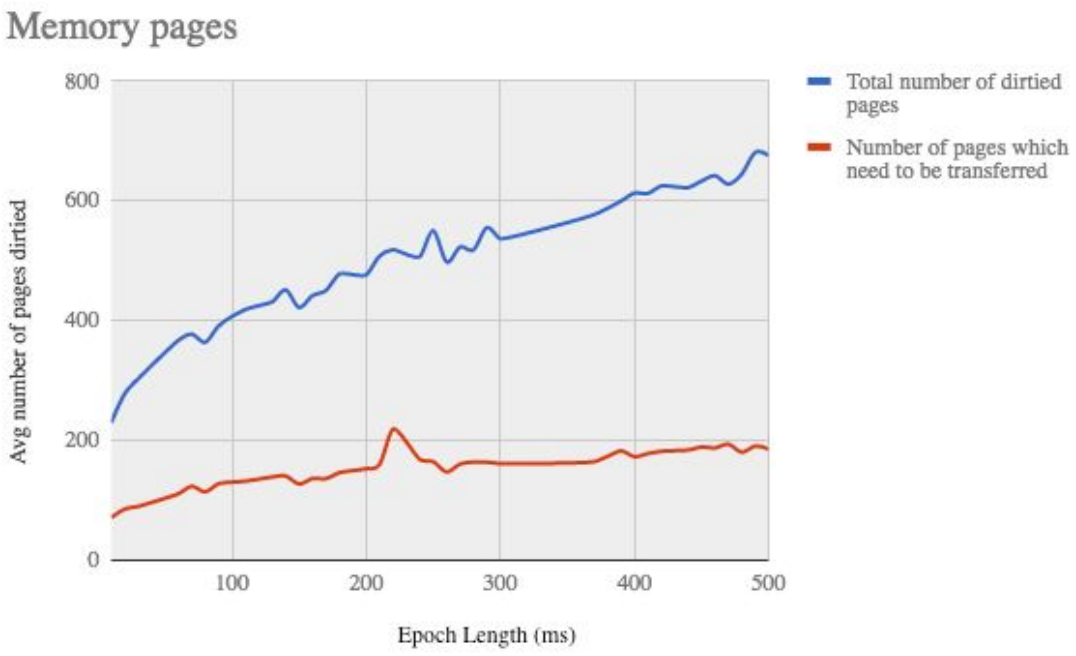

Figure 4.a FTP - Dirtied memory pages vs epoch length

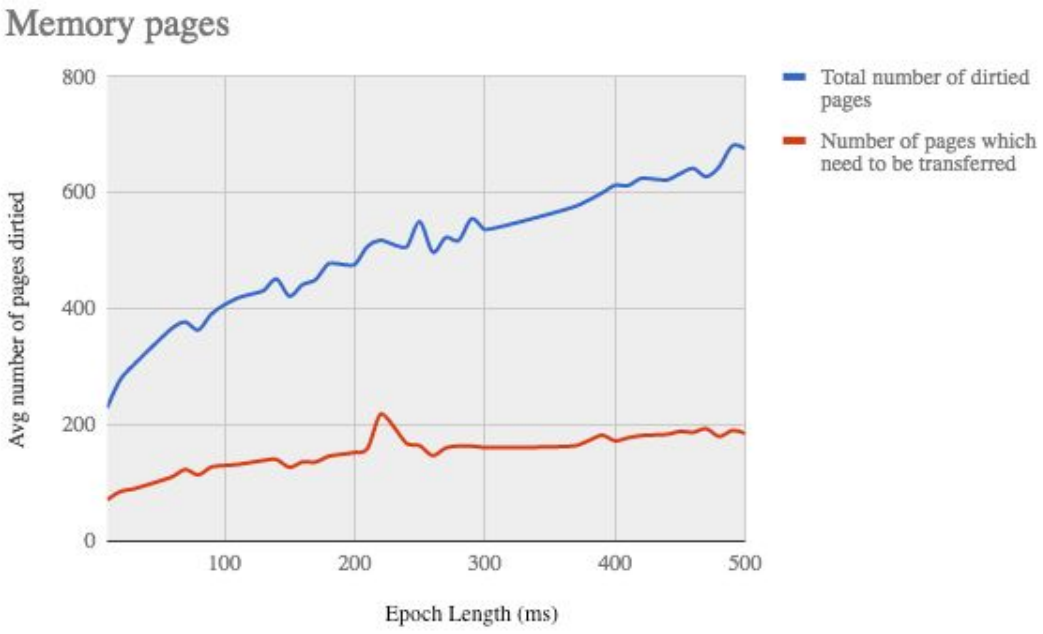

Figure 4.b FTP - Runtime vs epoch length

FTP is a network intensive application. It constantly sends and receives network packets. There is little processing involved. WIth increasing epoch length, runtime and the number of dirtied memory pages both increase. If we keep the epoch length at minimum, FTP will have a faster runtime.

e) NAS Parallel Benchmarks

NAS Parallel Benchmarks is a collection of computational intensive parallel applications performing various scientific computations developed by NASA [15]. Two of the benchmarks lu and sp were chosen for the tests.

Figure 5.a shows the number of dirtied memory pages in respect to the epoch length and Figure 5.b shows the runtime in respect to the epoch length. Due to the time consuming nature of this experiment, the experiment was performed only 12 times from 10ms to 500ms (with steps of 50ms).

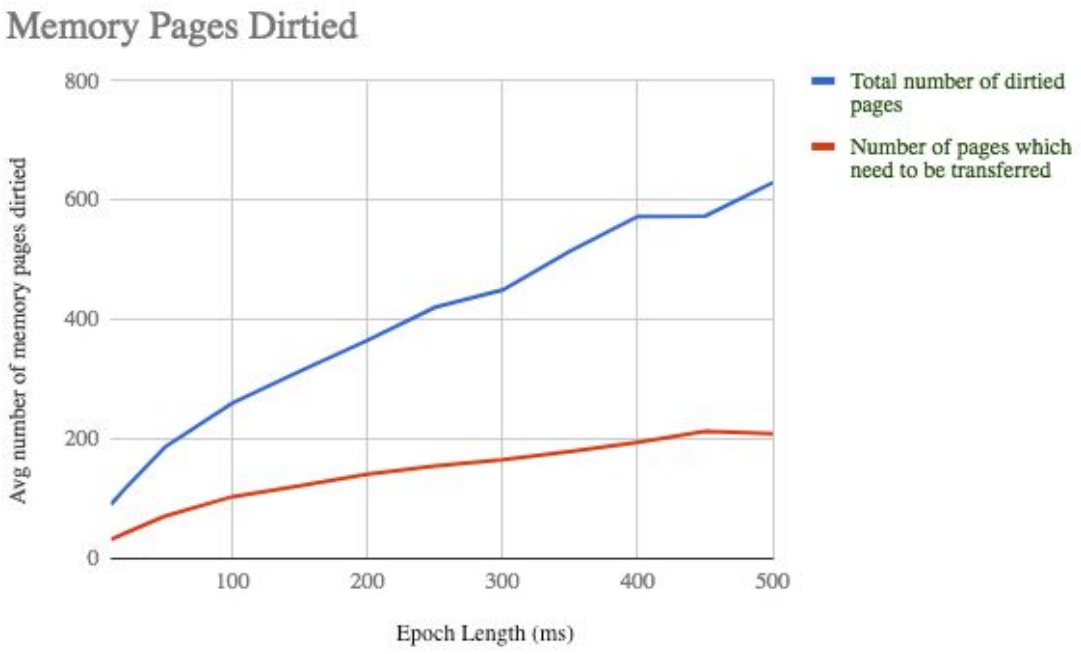

Figure 5.a NAS - Dirtied memory pages vs epoch length

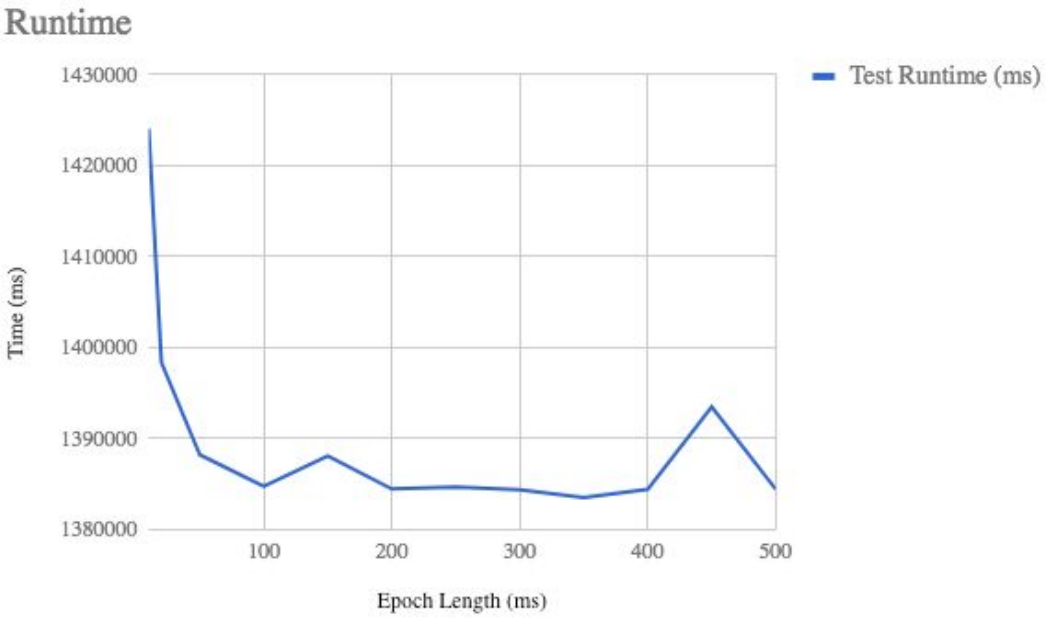

Figure 5.b NAS - Runtime vs epoch length

NAS parallel benchmarks did not require any network exchange and these experiments were very processing intensive. Although the number of memory pages being dirtied increase with increase in epoch length, the cumulative memory pages being sent are lower.

For example, let's consider epoch lengths of 10 ms and 100 ms. The system is transferring 315 memory pages in 100 ms with a checkpoint frequency of 10 ms. However, on the other hand the system only transfers 104 pages in 100ms.

Looking at Figure 5.b, we realise that the runtime actually decreases and then stabilizes as the epoch length is increased. This works in our favor. We can reduce the checkpoint frequency and at the same time we will be increasing the runtime.

Most of the times the Plover system ran with a checkpoint frequency of 10 ms even if it was doing no significant work. This is unnecessary as there is no network output flowing out, and hence there is no need for urgency to make the two systems synchronised.

V. Algorithm

The algorithm attempts to divide the running of applications on the system into two categories, networking mode and processing mode. The two modes are described below :-

i) Networking Mode (Default)

In this mode the network outgoing traffic is more than 0. In this mode the checkpointing is done as per the default PLOVER mode which is to wait for the guest to be in idle mode before carrying on with the checkpoint.

Calculating outgoing Network traffic at every checkpoint will create additional overhead which is not ideal in this mode. To avoid this additional overhead, Network outgoing traffic is only updated after a certain number of checkpoints. If the value of network outgoing traffic reaches zero, the algorithm would switch over to the processing mode.

ii) **Processing Mode**

This mode is activated when the network outgoing traffic is 0. In this mode the checkpointing is done after every 100ms to reduce the checkpointing overhead. Outgoing network traffic is updated after each checkpoint, and as soon as traffic is detected, the algorithm switches to Networking mode.

NOF	Network outgoing flow
MAX_CN	Maximum checkpoints in Networking mode before NOF is measured again
t	A discrete time instant
X	A counter to verify if MAX_CN is achieved or not

Table 1 - Variables used in the Algorithm

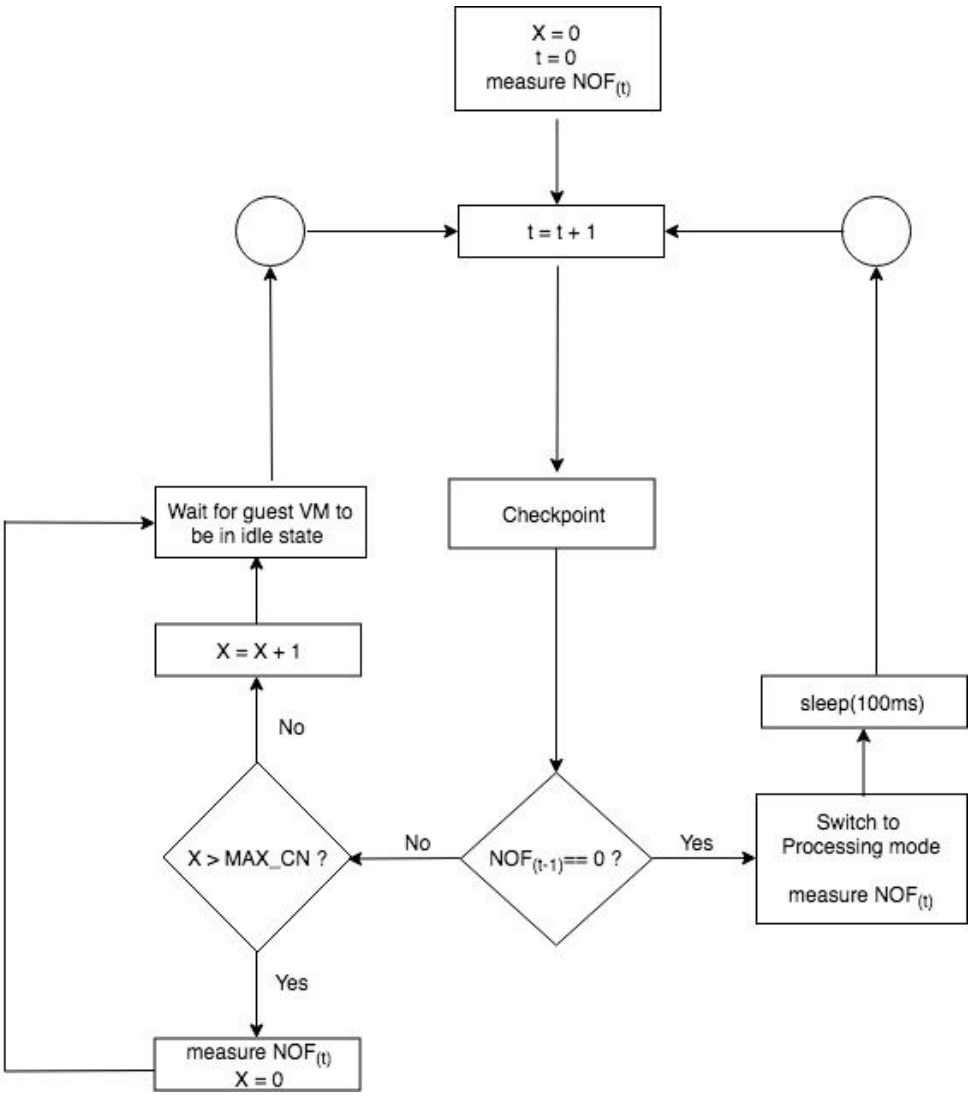

Figure 6 - The Algorithm

The Algorithm was then implemented using Plover. The code for the adaptive Plover program can be found at https://github.com/angad2102/Plover/tree/Adaptive-Plover.

Based on past research, epoch length of 100 ms was selected for processing mode [3]. MAX_CN was set to 75 so that the system waits for at least 750 milliseconds before trying to calculate the network output flow again. This number was derived from past experimental research [3].

VI. Evaluation

i) Experimental Setup

All the tests and experiments will be done on Dell R430 servers with Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. All NICs are Mellanox ConnectX-3 Pro 40Gbps connected with infiniband [10]. The ping latency between every two replicas is 84 microseconds (the TCP/IP over RDMA round-trip latency). They system is running Ubuntu 16.04.2.

The algorithm was tested on five programs, Mongoose, Apache (using PHP), Tomcat, FTP and NAS parallel benchmarks. The results were compared with that of Plover's running in its default mode.

ii) Results

Both Plover and and modified Plover with adaptive checkpointing gave similar results for Mongoose and Tomcat.

Plover with adapted checkpointing finished the Apache tests which were a combination of both processing and networking load in 13.651 seconds. Plover in default mode managed to complete the tests in 15.425 seconds. Plover with adaptive checkpointing performs better as it switches to

networking mode when Apache is receiving a request or sending out a web page and it switches to processing mode when Apache has to do some long calculations.

Performance of FTP was also similar in both Plover in default mode and Plover with adapted checkpointing.

Plover with adapted checkpointing finished the NAS parallel benchmarks set in 1384 seconds whereas Plover in default mode only managed to finish the set in 1424 seconds. The checkpoint frequency in Plover with adapted checkpointing was 100 ms whereas the checkpoint frequency in Plover in default mode was about 10 ms.

Overall the results for Plover with adaptive checkpointing are either better or equivalent to that of Plover running in default mode.

VII. Future Plans

In the future more work can be done on devising techniques to compress the data being transferred at each checkpoint and predicting page faults. The algorithm can also be integrated into more checkpoint based replication solutions.

VIII. Conclusion

Checkpoint-recovery based virtual machine replication is lucrative, providing high availability without modifying applications or using commodity hardware. However excessive checkpointing can cause significant overhead and reduce the system performance. The intention of this project was to improve the performance of such replication softwares using adaptive checkpointing.

Several tests were conducted on plover using five different applications and varying epoch lengths for checkpointing. These tests indicated that a lot of times, there is excessive checkpointing which can be avoided easily. This was especially true for applications which do not send out any network packets.

The algorithm described in this project achieves the goal of reducing overhead. The algorithm divides the workload into two group, network and processing. In network mode the algorithm executes checkpoints as they would have been executed in Plover whereas in processing mode the checkpoints are done at every 100 ms.

The algorithm shows promising results and is effectively able to reduce the runtime in processing intensive applications. The performance is also better in applications combining both processing and network communication. The algorithm's performance is the same as that of Plover in default mode for network intensive applications.

IX. References

[1] C. Wang, X. Chen, W. Jia, H. Qiu, S. Zhao, H. Cui, PLOVER: Fast and Scalable Virtual Machine Fault-tolerance on Multi-core, in Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI'18

[2] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, Remus: High availability via asynchronous virtual machine replication, in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI'08, USENIX Association, Berkeley, CA, USA, 2008, pp. 161–174

[3] B. Gerofi and Y. Ishikawa, Workload adaptive checkpoint scheduling of virtual machine replication, in IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC), Pasadena, CA, December 2011, pp. 204–213.Adaptive Remus

[4] Which Hardware Fails the Most and Why. http://www.storagecraft.com/blog/hardwarefailure/

[5] Strom, R.E. and Bacon, D.F. and Yemini, S.A., "Volatile logging in n-fault-tolerant distributed systems," in Fault-Tolerant Computing, Eighteenth International Symposium on, Jun 1988, pp. 44 –49.

[6] "What is a Virtual Machine and How Does it Work | Microsoft Azure", *Azure.microsoft.com*, 2017. [Online]. Available: https://azure.microsoft.com/en-in/overview/what-is-a-virtual-machine/.

[7] "10 Popular uses of Virtual Machines - MyTechLogy", *MyTechLogy*, 2017. https://www.mytechlogy.com/IT-blogs/9566/10-popular-uses-of-virtual-machines/#.We7e5xOCy8U.

[8] "What is high availability (HA)? - Definition from WhatIs.com", *SearchDataCenter*, 2017. http://searchdatacenter.techtarget.com/definition/high-availability.

[9] D. Sorin, *Fault tolerant computer architecture*. San Rafael, Calif.: Morgan & Claypool Publishers, 2009, p. 63.

[10] An Introduction to the InfiniBand Architecture. http://buyya.com/superstorage/chap42.pdf

[11] M. Lu and T. cker Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," in Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on, 2009, pp. 534 –543.

[12] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the Performance of Hypervisor-based Fault Tolerance," in Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1 –10.

[13] J. Hardman, "NAS Parallel Benchmarks", Nas.nasa.gov, 2017. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html. [Accessed: 01- Dec- 2017].

[14] Huawei FusionSphere. https://www. youtube.com/watch?v=yvsVuLAOhCo, 2014.

[15] "NASA. NAS parallel Benchmarks." http://www.nas.nasa.gov/Software/NPB