UNIVERSITY OF CALIFORNIA SAN DIEGO

Enabling Fast Recovery For Autonomous Vehicle Systems With Linux Container Checkpointing

A thesis submitted in partial satisfaction of the
requirements for the degree
Masters of Science

in

Electrical Engineering (Computer Engineering)

by

Maximilian Paulsen Apodaca

Committee in charge:

Professor Jishen Zhao, Chair
Professor Mohan Trivedi, Co-Chair
Professor Sujit Dey

2022

The thesis of Maximilian Paulsen Apodaca is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT OF THE THESIS

Enabling Fast Recovery For Autonomous Vehicle Systems
With Linux Container Checkpointing

by

Maximilian Paulsen Apodaca

Masters of Science in Electrical Engineering (Computer Engineering)

University of California San Diego 2022

Professor Jishen Zhao, Chair
Professor Mohan Trivedi, Co-Chair

Failures are unavoidable in engineered systems such as autonomous vehicles, but the latency of recovering a failed component degrades the performance of autonomous vehicles. We proposed a scheme to reduce the time of recovering autonomous vehicles from failures. Using Linux kernel features and container technology, we containerize functional components of autonomous vehicles and periodically take checkpoints. After detecting a failure, we recover the failed component to a previous state, and we notify the rest of the system to coordinate with the recovery. We test our method using the Robot Operating System (ROS), a widely-used middleware for robots and

autonomous driving vehicles. Our initial experimental results show that we reduced the recovery time of a practical pointcloud processing component in autonomous driving by 94%.

Chapter 1

Introduction

1.1 Introduction

The rapid evolution of autonomous driving technology has fostered the development of various intelligent and complicated software applications running on sophisticated server-grade systems [12](Figure 1.1). The trend towards high-automation levels like Level-4 and -5 which represent full autonomous control of the vehicle in most traffic situations create complexities. These complexity introduces a long latency to restart and recover from system and process failures. Without an efficient and fast recovery scheme, an AV may run with a downgraded performance, pull over until the full function of the vehicle recovers, or even temporarily lose the "driver" control in response to a process failure. These fallback mechanisms pose excessive safety hazards. In addition the remedies worsen the rider's experience and could require human intervention. This work aims to reduce the time to recover the full functionality of failed processes in AV systems at low run time cost.

We emphasize fast recovery of components in the primary control pipeline; other safety strategies, such as fallback controllers outlined by Ishigooka et al. [9] or redundant backup systems, can co-exist with fast recovery schemes to further improve AV robustness and safety.

1.2 Challenges

Fast AV system recovery prompts substantial system design challenges. First, it is infeasible for AV systems to directly adopt fault tolerance and recovery methods used in traditional servers and mission-critical systems. Compared to conventional servers, AV systems have limited redundancy, power, thermal budgets, and form factor [12]; it is impractical to adopt the existing system duplication and data replication used in server clusters. It is also impractical for a commercial AV to employ expensive safe island solutions used in air/space crafts. Secondly, the modern AV software stack consists of many software modules running on a combination of Linux and robotics operating systems (ROS) [12]. Although recent studies, e.g., Rorg [19], adapt Linux containers to orchestrate the system resource management of Linux and ROS, fast recovery of application processes remains largely unexplored.

1.3 Proposed approach

To address the challenges, we propose a fast AV system recovery scheme by exploiting Linux kernel features such as copy on write pages and container technologies. Our key idea is to isolate autonomous driving software modules as self-contained containers and perform checkpointing and restore at the container level. After detecting a failure, we recover the failed component to a previous state using the checkpoints and notify the rest of the system to coordinate with the recovery. Our evaluation demonstrates that our design effectively reduces the recovery time of practical AV applications, such as pointcloud processing, by 94%.

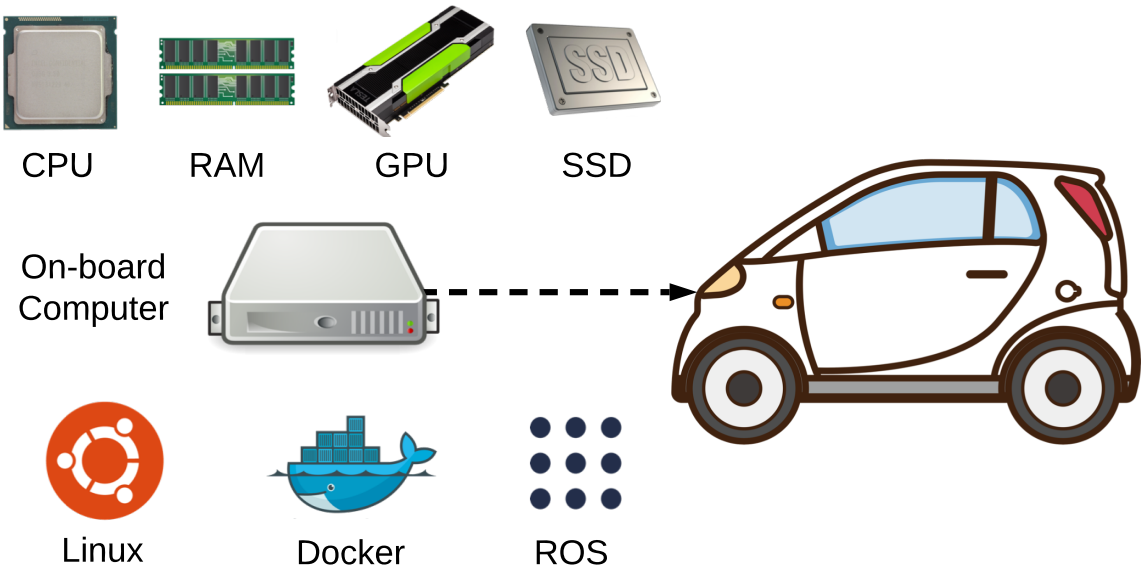
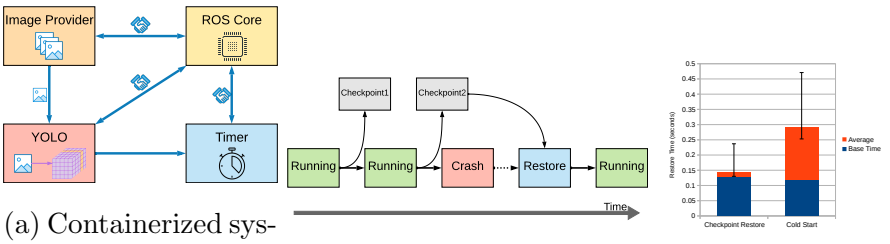


Figure 1.1: The software/hardware components of an autonomous driving system. Today, a Level-4 autonomous driving system adopts server-grade software and hardware to accommodate the intensive computation and storage requirements in AV applications.



(a) Containerized system

(b) Checkpoint process

(c) Recovery Time

Figure 1.2: (a) Communication of ROS processes in individual containers. (b) Periodic checkpointing of containers and a restore after a crash. (c) A comparison between a cold start and checkpoint/restore for the voxel_grid_filter Autoware Node [12].

Chapter 2

Background and Related Work

Recovery strategies for autonomous vehicles must restore the state of an autonomous vehicle if a software fault is detected. This is important in modern high-automation AV system. In addition, most modern high-automation AV systems are built on Linux operating systems and robotic middleware such as ROS [12]. As such we can use Linux kernel features to aid in the fault tolerance of applications. For instance applications communicate through common network protocols such as TCP and usually have an option to run in shared memory mode which would allow us to provide clean abstractions for checkpointing. In addition, the software which autonomous vehicles run does not change often which allows us to optimize checkpointing approaches. An ideal solution would provide flexibility for programmers while requiring little additional hardware.

Most previous fast recovery schemes focused on kernel level designs that recover from device driver [16] and OS crashes [5]; there were also attempts to build an operating system that automatically recovers from failures [6]. However, vast majority of robotic software components, especially ROS-based software, execute in the user space. Previous kernel-level recovery mechanisms can impose unnecessary recovery latency and complexity upon user process failures. Alternatively, user-space checkpointing mechanisms, such as Distributed MultiThreaded CheckPointing (DMTCP) [2] and Checkpoint/Restore In Userspace (CRIU) [3], are often geared towards datacenter checkpointing where checkpoint latency is much more important than restore

latency. We examine these drawbacks in later sections.

2.1 DMTCP, CRIU, and Derivatives

DMTCP and CRIU work in similar manners. They both freeze a target process, write its state to storage, and then optionally resume the process execution. The data saved include the pages of process memory, TCP / UDP connections established, and file descriptors open among other metadata. The difference between DMTCP and CRIU is that DMTCP uses a user space library to execute code in the process space while CRIU uses the ptrace API to inject its parasite. Previous work attempted to incorporate DMTCP into error recovery for ROS [2]. However, most of these mechanisms involve checkpointing the whole ROS stack to recover from a failure in the ROS master node. Checkpointing the whole ROS stack simplifies recovery but also requires every node to wait for the whole checkpointing process before resuming execution. This introduces large delays in checkpointing time [10].

Linux containers are becoming a mainstream way to deploy software in robots. Recent works, such as Rorg [19], employ Linux containers to run ROS programs to reduce maintenance burden and resource contention. In this work, we leverage Linux container's encapsulation to checkpoint and restore standalone user space application components. Employing containers is practical since ROS applications can communicate over a network based protocol. The network communication allows us to decrease the number of applications that need to be checkedpointed at the same time. We leverage the existing boundaries created by the TCP connections by separating each application into its own container. Container level checkpointing is a common level of abstraction for data-center checkpoint restore schemes such as CRIU, HyCoR [21] and NiLiCon [20] which all rely on containers to provide an interface to checkpoint, i.e. the whole container is checkpointed at once. In fact, Docker, a popular container management system, uses CRIU for its built-in checkpoint restore mechanism.

General purpose container checkpoint and restore has been demonstrated to work with sub 10 millisecond recovery time [21]. The highest performing solution, HyCoR, requires a second server to host a backup container, increasing resource demands on

the system. NiLiCon, which HyCoR is based on, does not require the second server but has restore times over 100 milliseconds. NiLiCon and HyCoR both use CRIU as a backend to track pages necessary to save for checkpointing in addition to the other process state such as TCP connections and open file descriptors. Other proposed cold backup solutions require over 100 milliseconds to recover [18].

2.2 Other Approaches

Another approach is to use hand crafted checkpoint restore mechanisms. This is very common in high performance compute applications and neural network training. [15]. Hand crafted solutions require expertise from the software developer and modification of the software to meet requirements by the developers. There exist libraries that assist in the writing of hand crafted checkpoint restore code such as [8]. This provides additional features such as code modification in between checkpoints. That is the software running on the autonomous vehicle is modified between a checkpoint restore cycle. We do not explore this line of work and instead focus on checkpointing mechanisms leveraging existing abstractions created by containers are described earlier.

2.3 Extensions

Previous work examined checkpoint restore for autonomous vehicles from a systems perspective by determining the optimal checkpoint frequency [13]. We do not address the checkpoint frequency in this work but delegate to Kim et all's work instead.

Many common autonomous vehcicle applications require the use of GPUs for efficient computation. These pose unique challenges when running checkpoint restore as the memory is fragmented across two different devices. One solution is proposed in CRAC [11]. CRAC substitutes the GPU driver library, the CUDA library, with a custom version that tracks dirty pages. The dirty page tracking is then used to copy necessary data to the CPU on checkpoint and restore. These are additional

requirements in regards to the shared VMA implementation in CUDA which are described in Twinkle's paper. In this scope we focus on only checkpoint and restore of CPU processes but acknowledge the importance of extending the work to GPU processes.

2.4 Limitations

This work tackles reliability of autonomous vehicles, however there are many aspects of reliability which are not address by this work. We focus on user level application resilience from the perspective of intermittent issues. Other work address topics such as network resiliency [17] or other algorithmic and kernel level resiliency such as recovering from device driver [16] and OS crashes [5].

Chapter 3

Design and Implementation

We propose a scheme to perform periodic checkpointing of containerized ROS applications during regular operation with the ability to fast recover upon failures. To demonstrate our design, we implement our system on Linux and ROS.

3.1 Goal

Fast recovery involves restoring a process to a previous state in the presence of an intermittent software error. For instance a bug that only occurs at a particular time or in response to a particular packet. An ideal solution would skip over the bug inducing time step or sensor data.

To achieve our goal of restoring a failed application process as quickly as possible, we opt to employ periodic checkpointing. Every interval (a few seconds based the profiling of our system), we take a snapshot of the ROS node. If we encounter a process error or failure, we restore from the previous checkpoint. Figure 1.2b illustrates the checkpoint restore process, while Figure 1.2a shows an example containerized ROS system.

3.2 Naive Implementation

A naive implementation could use off the shelf container implementations. Many containerization frameworks support checkpoint restore [7] of containers. These often use CRIU which saves the program state to disk. There is a large amount of overhead associated with writing data to disk [18] which limits the feasibility of this approach. In addition, standard checkpoint restore schemes with CRIU are unable to checkpoint established TCP connections. That is to say we can not checkpoint a TCP connection to another container.

Our proposed scheme leverages the copy on write mechanism in the Linux kernel commonly used by the clone system call to overcome the overhead associated with writing data to disk. Similar to VAS-CRIU [18] this places the checkpoint data into volatile memory. However we significantly reduce overhead by employing the copy on write scheme. The copy on write mechanism in the Linux kernel copies a page of memory only when it is written to. This means static pages between checkpoints are not written to disk. There is a performance overhead associated with this while the application is running, however we found this to be negligible as described in section 4.3.

Our scheme also handles TCP connections which are the main mechanism of cross container communication. The scheme does not create a new TCP connection, instead it re-uses the existing connection. This has the side effect of keeping the queues persistent across checkpoints. That is to say the packets received after a checkpoint but before a restore are effectively dropped. This has effects on stability but the scheme as a whole manages high reliability as described in section 4.4.

3.3 Background on Linux Processes

To persist TCP connections and use the copy on write mechanism featured in the Linux kernel we clone a process. In Linux each process has a parent and can optionally share the address space with its parent. In the case of a process fork the address spaces are shared which means a write from the child will affect the memory

of the parent and vice versa. However, in the case of a clone the address spaces for the child and the parent are separate but the file descriptors can persist across both processes. Each time a process invokes fork or clone a new kernel thread is created for the process. This is a process with a parent that optionally share the address space with its parent process. This means that each process running in the container can, and often does, have many child processes acting as threads.

3.4 Cloning a Process

Each time we want to make a checkpoint we invoke the clone syscall from the desired process. We then walk the process tree for the container to build the kernel threads for each process. This results in two identical process trees which are backed by the same physical pages. However, the kernel is set up to perform a copy on write for each tree's address space. During this process we freeze each of the processes in the process tree, then after completing the clone we release the last process and allow normal execution to resume. At the same time we keep the cloned tree frozen. When the original tree crashes or encounters an error we unfreeze the cloned tree and kill the original tree. This approach performs much fast checkpoint restore operations providing sub 100 ms checkpoint and sub 100 ms restore times on the same system with minimal performance degradation.

To duplicate the address space we employ a parasite. This is a short executable which we inject into the target's address space by means of the ptrace kernel API. The compel program provided by CRIU is used heavily in this process. Compel provides facilities to inflect and cure the target process with a user provided parasite. Infect is used to describe mapping the parasite executable into the victim's address space. Unmapping those pages is referred to as curing. Figure 3.1 shows the detailed clone process.

When we go to do a checkpoint we first walk the process tree while the application is still running. We can do this as the autonomous vehicle applications are fairly static in their process tree. After this we have a tree of processes to duplicate. We then freeze the process tree and infect it with our parasite. This involves using the ptrace API to

allocate pages for our parasite and copy the executable. The parasite is invoked with the process tree and calls the clone syscall in the appropriate manner as to duplicate the process tree. This involves one call to clone that does not preserve the address space and instead creates a copy on write copy. After this we free the original process tree and also resume execution of the parasite in the cloned process tree. Here we duplicate the process tree by calling clone and specifying that the address space is to remain the same. At the end we also cure the cloned process tree. This yields two identical process trees, one of which is executing and the other of which is frozen. To do a restore we must simply resume execution on the cloned process tree which is a single syscall per fork.

3.5 Parasite Operation

The parasite is responsible for executing the clone syscall in the victim process and the duplicated process tree. A parasite is a small program which we run in the address space of another process. In this case we employ two parasites. The first is responsible for duplicating the address space from our running process. This involves a single call to the clone syscall with the appropriate parameters. The rest of the process creation is done in the cloned process tree. We start by using the compel library to freeze the victim parent process by means of the ptrace API. This allows us to write to memory locations in the victims address space. We map the parasite into the victims memory as described by the compel documentation [3] . The executable we map invokes the clone syscall via some inline assembly code. It is important not to use any glibc functions as the parasite should not interfere with any memory of the victim process. After the parasite executes the clone syscall the address space is duplicated. This sends a signal to the initiator of the ptrace calls and allows us to capture the child process ID which corresponds to the cloned syscall. These steps are illustrated in blocks b and c in figure 3.1.

After receiving the child process ID we no longer need the parasite in the victim so we remove the parasite from the victim. Now we need to create the threads present in the original process tree in the cloned process tree. This requires us to attach

another parasite but this time to the parent thread of the cloned process tree. Using the parasite we call the clone system call repeatedly for each of the various threads found in the original process tree. Each time we call the clone syscall we make sure to unfreeze the parasite but keep the new process frozen. This means we never let the cloned process tree execute while the original process tree is running. Once again we record the PID of each thread. After we finish creating the threads we are done will all the parasite and unmap it from the cloned address space. This process is illustrated in figure 3.1.

3.6 Metadata Collection

To restore the process we need to know what each thread in the process was executing. This way we can restore the stack pointer and program counter to the correct locations. We gather the metadata using the ptrace API. Freezing a process yields the necessary data present. We freeze each of the threads of the process to checkpoint and gather the metadata from each thread. This adds significant overhead as we have to freeze each child process individually which greatly increases the total time the process spends frozen. We found little difference in recovery percentage depending on if we froze all the threads at the same time or one after each other. As a result we do not freeze all threads at the same time but freeze one at a time. See section 4.3 for more information about the overhead and section 4.4 for more information about restore success rates.

3.7 Interaction within Containers

The basis for our checkpoint restore scheme is the abstraction given by containerizing each application. We use podman [4] as the containerization software, however there is nothing inherent to podman that could not be replicated in other software. The key functionality offered by containers is that we can isolate the applications to checkpoint and guarantee that they only communicate over network connections. This allows us to simply walk the process tree found in a container to determine the

processes we need to checkpoint and restore in one block.

3.8 Reliability

Another aspect of checkpoint restore is the ability to successfully complete a restore. Depending on the implementation a restore might fail a certain percent of the time. This could be due to a poorly chosen checkpoint location or an inconsistent checkpoint state. For example Yu Huang et all's work, which has the ability to restore different versions of the software, does not feature a 100% restore rate, especially when updating the software at the same time [8].

The most common failures we encountered in testing were the result of non-standard error messages. Some syscalls relating to futexs do not fail under normal operation, however after being frozen by ptrace the syscall could return with the error code EAGAIN [1]. This indicates to the program that nothing is wrong but the call should be attempted again, however the program do not always respect this error code which sometimes leads to unrecoverable errors. We only noticed these errors in the restore stage and never in the checkpoint stage.

3.9 Summary

Our solution have similar pitfalls which means the restore process does not succeed every time. This contrasts to other solutions which provide guaranteed restores. We make this trade-off in the name of performance. The details of the restore success rates are discussed in the evaluation.

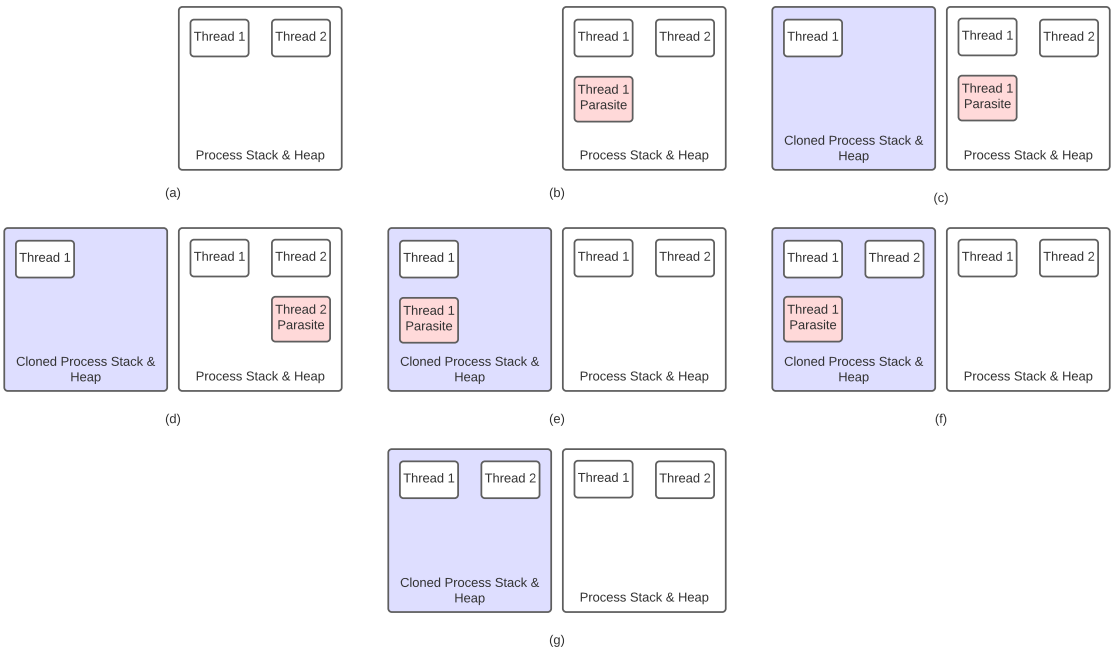


Figure 3.1: The steps to checkpoint a process. (a) Before checkpoint. (b) The parasite attaches. (c) Duplicate the address space. (d) Acquire thread 2 metadata. (e) Attach to the clone. (f) Create a second cloned thread. (g) The completed checkpoint process.

Chapter 4

Evaluation

We evaluated our work based on the Autoware self driving system. This is an open source autonomous vehicle software stack based on ROS. It consists of processes described as nodes which communicate with each other via TCP connections. Each node is isolated in its own container as part of our system, this is representative of organization schemes such as RORG [19]. There are generally two types of nodes, GPU nodes and CPU nodes. GPU nodes require a GPU and are excluded in our implementation, in this case they are not checkpoitned and instead continue running without interruption. We examine a subset of the CPU nodes.

4.1 System Setup

All of the experiments are run on a system with an Intel i7-4700MQ CPU, 16 GB memory, a SATA SSD, Ubuntu 18.04 with kernel 4.15.0-147-generic and ROS melodic release, similar to the system used by our autonomous vehicles [14]. We ran each experiment twenty times to gather averages and standard deviations for each value. Each of the experiments were conducted by replaying a ROS Bag of a driving scenario. This was done to reduce run to run variance.

4.2 Restore Overhead

To determine checkpoint overhead we examine the delay caused by each checkpoint operation. This is measured as an end to end delay for each processing chain. For instance the point down-sampler node receives LiDAR data and outputs point cloud data to the NDT matching node. The NDT matching node produces a position. We measure the added delay measured at the output of the NDT matching node. For instance the expected delay between position updates is 100 ms. If we measure this delay to be 110 ms after a checkpoint operation which indicates a 10 ms overhead.

To give a sense of how our design would perform in a realistic scenario we evaluated design on Autoware's perception stack [12]. A timer measured the interval between processed messages which gives us an upper bound on the end to end system latency introduced by our solution. During the measurements we checkpointed the voxel grid filter node which downsamples incoming LiDAR data. We observed a 82% (107ms) increase in maximum end to end latency after a checkpoint restore cycle compared to the 389% (471 ms) increase of a cold start. In the median case we observe 12% (15ms) and 241% (292ms) increases respectively. This compares favorably to the previous single node checkpoint restore solutions that take on the order of 100 milliseconds and is in the same range as multi node solutions.

In addition, we measure the restore success. In practice we found it unnecessary to measure checkpoint success since we did not encounter a single failure in the checkpoint creation process. The application fails to restore if the complete chain of nodes fails to output a result. For instance if we restore the failed node and it successfully establishes a connection with the upstream node but not the downstream node then this is a failure. A success would be if the full chain successfully processes data. In the NDT / Voxel downsampler example this would be a successful chain from the bag, to the voxel downsample node, to the NDT matching node and finally the measurement node.

4.3 Runtime Overhead

Finally we need to measure the checkpoint creation overhead. We break the overhead down into three parts. The memory footprint increase, as in how much additional memory we require. This is straight forward to measure. Then the freeze time or the time during which the process is unresponsive. We measure this in a similar manner to the restore overhead in that we dtermin the time during which the process to checkpoint breaks the computation chain. Finally, since we use a copy on write scheme we need to measure the performance overhead of the copy on write operations. We examine the additional memory overhead and CPU usage as shown in figure 4.1. There is an increase memory usage of 5% and no measurable increase in CPU usage during the time. That is to say that it was below 4% of a core.

4.4 Restore Failures

During the restore process we see some failures. These failures generally fall into one of a few categories. Most of the failures revolve around an inconsistent state for Futex's. This is because the code used to wait for a Futex does not interpret the error code for try again correctly. This is because in normal execution this error number does not get set. However, when interrupting the Futex in the fork the restore does not behave correctly. This causes errors within ROS. However these errors are found to occur infrequently.

When examining the effectiveness of this checkpointing scheme it is important to keep in mind a few key considerations. The first is that the overhead of the scheme is very low. The second is that the alternative to this scheme is another scheme or to fall back to a failure. And finally, the scheme does not rely on any code modification. The scheme overhead is low enough to not interfere with the operation of most software. This makes it possible to run the software on all useful nodes. This increases the usefulness of the software in a real world deployment when compared to other software systems which require a backup system or have a larger performance overhead. The second consideration is that the comparison for the scheme should

not be a perfect backup system but instead one of the other backup schemes. This is because the scheme is fast enough that another scheme can be applied afterwards in the event of a failure. This means that any improvement upon the base case of a cold start is an improvement. An alternative backup system could improve this rate farther but would work in addition to our system for negligible performance implications.

4.5 Conclusion

The final point is that the scheme does not require any code modifications. This is a key advantage as compared to other checkpoint restore mechanisms. Without code modification the same scheme can be used for independent software which must be brought together or used in a black box fashion. This enables integration with vendor software that does not use the scheme or other tools that make it challenging to implement.

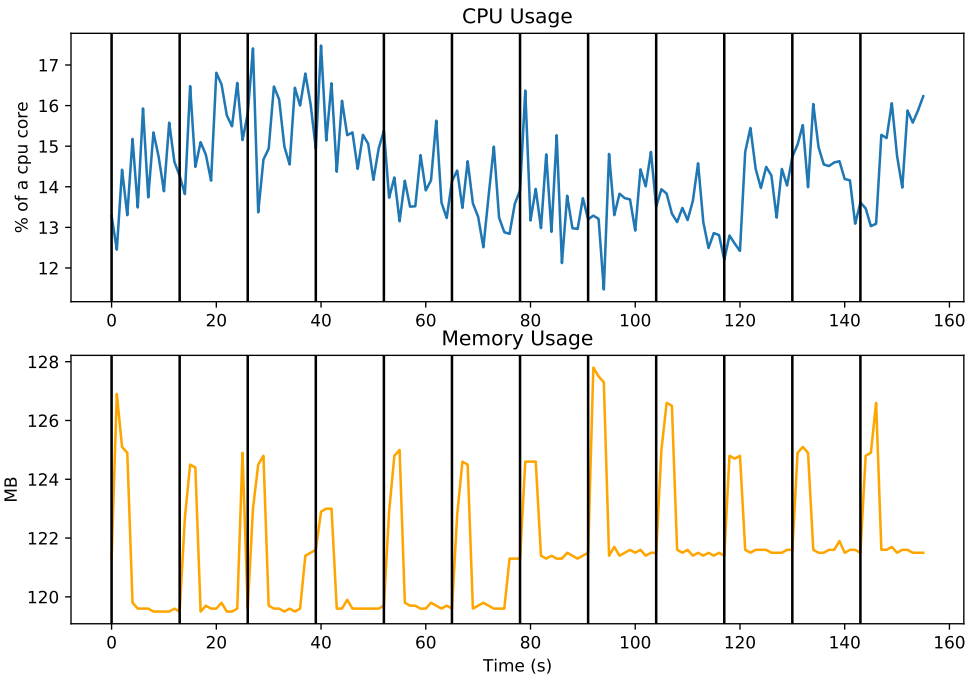


Figure 4.1: CPU and Memory overhead after repeated checkpoint cycles. Each checkpoint is kept for three seconds. Black lines indicate the checkpoint time.

Chapter 5

Conclusion

Failures are unavoidable in engineered systems such as autonomous vehicles, but the latency of recovering a failed component degrades the performance of autonomous vehicles. This paper proposed a scheme to reduce the time of recovering autonomous vehicles from failures using Linux containers and checkpoints. Existing work in checkpointing showed fast recovery however required additional backup systems. We accomplish this by utilizing autonomous vehicle specific checkpoint grantees, such as the grantee that the checkpoint and restore systems are the same machine. Experimental results show a 94% reduction in recovery time for a realistic autonomous driving workload with minimal performance overhead. The scheme does not require additional hardware and works on off the shelf software running in containers.

When compared to the alternative of not using a checkpoint restore system our work is able to recover from errors which could leave the car non-operational. The existing mechanisms do not provide sufficient performance or require large amounts of resources. The current work shows promise for real world scenarios, however does not checkpoint CUDA / GPU processes. While this is a limitation, it does not prevent the deployment of this work as the work can be applied on a container by container basis to CPU processes.

5.1 Future Work

Our future goal is to apply this technique to more real-world AV scenarios and further improve the robustness of autonomous driving. In addition, we aim to explore the check pointing and restoration of CUDA applications to allow greater coverage of AV software.

Bibliography

[1] *errno(3) Linux User's Manual*, Feb 2022.

[2] J. Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *IPDPS*, 2009.

[3] CRIU community. Checkpoint/restart in userspace(criu), 2021.

[4] Containers. Podman is a daemonless container engine for developing, managing, and running oci containers on your linux system., February 2022.

[5] Francis David and Daniel Chen. Recovering from operating system crashes. 2008.

[6] Francis M. David and Roy H. Campbell. Building a self-healing operating system. In *DASC*, 2007.

[7] Docker Inc. Docker is an open platform to build, ship and run distributed applications anywhere., November 2018.

[8] Yu Huang, Kevin Angstadt, Kevin Leach, and Westley Weimer. Selective symbolic type-guided checkpointing and restoration for autonomous vehicle repair. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 3–10, 2020.

[9] T. Ishigooka, Shinya Honda, and Hiroaki Takada. Cost-effective redundancy approach for fail-operational autonomous driving system. In *ISORC*, 2018.

[10] Twinkle Jain and Gene Cooperman. Dmtcp: Fixing the single point of failure of the ros master, 2017.

[11] Twinkle Jain and Gene Cooperman. Crac: Checkpoint-restart architecture for cuda with streams and uvm. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[12] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An open approach to autonomous vehicles. *IEEE Micro*, 35(6), 2015.

[13] Taesik Kim, Hong Min, and Jinman Jung. Vehicular datacenter modeling for cloud computing: Considering capacity and leave rate of vehicles. *Future Generation Computer Systems*, 88:363–372, 2018.

[14] David Paz, Po-Jung Lai, Sumukha Harish, Hengyuan Zhang, Nathan Chan, Chun Hu, Sumit Binnani, and Henrik Christensen. Lessons learned from deploying autonomous vehicles at UC San Diego. In *FSR*, 2019.

[15] Elvis Rojas, Albert Njoroge Kahira, Esteban Meneses, Leonardo Bautista Gomez, and Rosa M Badia. A study of checkpointing in large scale training of deep neural networks. *arXiv preprint arXiv:2012.00825*, 2020.

[16] Michael M. Swift, Muthukaruppan Annamalia, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, November 2006.

[17] Shane Tuohy, Martin Glavin, Ciarán Hughes, Edward Jones, Mohan Trivedi, and Liam Kilmartin. Intra-vehicle networks: A review. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):534–545, 2015.

[18] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *MEMSYS '19: Proceedings of the International Symposium on Memory Systems*, pages 53–65, 2019.

[19] Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen. Rorg: Service robot software management with linux containers. In *ICRA*, 2019.

[20] Diyu Zhou and Yuval Tamir. Fault-tolerant containers using nilicon. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1082–1091, 2020.

[21] Diyu Zhou and Yuval Tamir. Hycor: Fault-tolerant replicated containers based on checkpoint and replay. *CoRR*, abs/2101.09584, 2021.