Autonomy Today: Many Delay-Prone Black Boxes

Sizhe Liu
University of North Carolina at Chapel Hill, NC, USA

Rohan Wagle
University of North Carolina at Chapel Hill, NC, USA

James H. Anderson
University of North Carolina at Chapel Hill, NC, USA

Ming Yang
WeRide Corp., San Jose, CA, USA

Chi Zhang
WeRide Corp., San Jose, CA, USA

Yunhua Li
WeRide Corp., San Jose, CA, USA

Abstract

Machine-learning (ML) technology has been a key enabler in the push towards realizing ever more sophisticated autonomous-driving features. In deploying such technology, the automotive industry has relied heavily on using "black-box" software and hardware components that were originally intended for non-safety-critical contexts, without a full understanding of their real-time capabilities. A prime example of such a component is CUDA, which is fundamental to the acceleration of ML algorithms using NVIDIA GPUs. In this paper, evidence is presented demonstrating that CUDA can cause unbounded task delays. Such delays are the result of CUDA's usage of synchronization mechanisms in the POSIX thread (pthread) library, so the latter is implicated as a delay-prone component as well. Such synchronization delays are shown to be the source of a system failure that occurred in an actual autonomous vehicle system during testing at WeRide. Motivated by these findings, a broader experimental study is presented that demonstrates several real-time deficiencies in CUDA, the glibc pthread library, Linux, and the POSIX interface of the safety-certified QNX Operating System for Safety. Partial mitigations for these deficiencies are presented and further actions are proposed for real-time researchers and developers to integrate more complete mitigations.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Software and its engineering → Process synchronization

Keywords and phrases autonomous driving, CUDA programming, locking protocols, POSIX thread, operating systems, machine learning systems, real-time systems

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.12

Supplementary Material *Software (ECRTS 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.1.3`

Software (Source Code): `https://github.com/sizheliu-unc/ECRTS24` [29]
archived at `swh:1:dir:825b348c28207aef045b6565ae0e977add42d44c`

Funding *Sizhe Liu, Rohan Wagle, and James H. Anderson:* supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, and CPS 2333120.

Acknowledgements We thank Huazhong Ning at WeRide for coordinating this collaboration and BlackBerry QNX for supplying relevant software.

1 Introduction

Graphics processing units (GPUs) have been a key technology in enabling advances in machine learning (ML) generally, and deep learning (DL) specifically. The utility of GPUs in this regard stems from their ability to accelerate complex ML/DL operations that would otherwise be performance bottlenecks. Currently, there is much interest in applying DL algorithms in ever more sophisticated, time-sensitive contexts. Autonomous vehicles (AVs) are a good exemplar where many challenges exist: here, fast processing most of the time is not good enough, as any processing delay could potentially cause an accident.

One would hope that, with sufficiently powerful hardware accelerators, such delays would be a non-issue. Unfortunately, accelerators are subject to contention-related overheads that are problematic in building real-time DL applications such as AV software. AV systems often consist of tens to hundreds of DL models of varying sizes with complex dependency relations [4, 32]. Even seemingly innocuous delays in this complex web of concurrently executed DL models could potentially cause system-level latencies with safety ramifications.

Focus of this paper. This paper is directed at studying unexplained CPU-side delays observed when launching GPU-accelerated DL model inference functions in a real-world AV system under test at WeRide.[1] The software in question was built with the NVIDIA CUDA API, a popular framework for programming GPU-accelerated applications, including DL inference. In vehicle testing, certain launches of CUDA[2] kernels (*i.e.*, DL models) experienced latencies of around $2\,ms$, as opposed to the expected $30\,\mu s$ in normal cases. These abnormal CUDA kernel launch latencies caused the vehicle to stall and revert to its backup module.

In further experimentation involving synthetic scenarios, we found that kernel launches can be delayed for over $100\,ms$, or even become blocked indefinitely. For reference, we found that 224×224 ViT-S, a state-of-the-art computer-vision model, takes $2\,ms$ to perform an inference on an NVIDIA Titan V GPU.[3] Thus, our observation of $2\,ms$ CPU-side kernel launch delays implies that initiating the DL model alone could take as long as the inference execution itself. This significantly complicates timing and schedulability analysis of the system, which consequently could lead to accidents in the context of an AV. This paper details our investigation of these delays, revealing their effects across various hardware and software configurations, and ultimately uncovers their root cause.

The high launch latencies first appeared to be a CUDA-specific problem, but our investigation ultimately revealed that the latencies stem from not only CUDA but also deep-rooted problems in the C Standard Library (LIBC) and the Linux kernel. For example, we found that the Linux GNU C Library (glibc) read-write lock implementation contains a critical flaw that can cause high latencies. Our findings also show that safety-certified operating systems (OSes), most notably the QNX OS for Safety ("QNX" for short), which CUDA supports,[4] are not immune to such lock-contention delays. In this paper, we discuss how deficiencies in these software components can ultimately contribute to delays in launching DL models. Although this paper primarily focuses on safety-critical AV contexts, our findings may apply to other real-time systems that rely on such software as CUDA, LIBC, and Linux.

---

[1] `https://www.weride.ai`
[2] Data is based on experiments with CUDA 11.2.
[3] Inference latency is also dependent on the ML compiler and implementation. In this experiment, we used a publicly accessible ViT-S ONNX implementation and TensorRT.
[4] Currently, QNX is the only safety-certified OS supported by CUDA.

So many black boxes. Our findings have relevance to the broader problem of developers treating foundational software components such as CUDA, glibc, and Linux as "black boxes."[5] These components serve a fundamental role in building not only safety-critical AV systems, but also a significant portion of software today. Despite their ubiquity in modern software development, their implementation details are seldom inspected thoroughly by their users, including AV developers. This is not without reason: these components are large, complex code bases that have proven sufficient across a vast range of use cases. A thorough audit of these sometimes opaque and undocumented implementations would require significant time and labor. Although the real-time systems community has identified many limitations in these components with respect to real-time functionality [7, pg. 157], [22, 45, 48], they are still deployed in real-time systems without mitigation. While this black-box approach to system design is by no means new, its increasing usage in the development of safety-critical systems, such as AVs, warrants scrutiny, as they are deployed in high-stakes scenarios.

Contributions. This paper reports on an investigation into significant delays in key CUDA functions on a real AV that resulted in problematic system-wise anomalies. Such delays in an AV can result in catastrophic outcomes, including casualties. This paper aims to identify and mitigate these delays, and provides tools and techniques that can be employed to address similar problems. The major contributions of our investigation are threefold.

First, by investigating the delays in our AV via system tracing experiments and source-code analysis, we reveal that CUDA relies on glibc read-write locks when making asynchronous CUDA kernel launches. We further show that the read-write lock code in glibc 2.23 has a critical implementation flaw[6] that is the root cause of the significant delays seen in CUDA in our AV. Expanding on these findings, we conducted additional experiments on the most recent version of CUDA (12.2.2) and glibc (2.38) in both discrete and unified (Tegra) NVIDIA GPU architectures to study if they can still result in extensive delays. Our findings reveal two commonly used CUDA functions to be susceptible to contention and priority-inversion that lead to extensive delays. This is because they heavily use mutex and read-write locks that lack proper real-time progress mechanisms. We show that CUDA 12.2.2 and the safety-certified QNX are not immune to these issues.

Second, we explain in detail the tools and techniques we used in investigating the black-box software components of relevance to our investigation, providing generalizable insights into addressing similar issues in other black-box components. We show the extent of the observed CPU-side delay by profiling a system of common DL models under contention and provide detailed locking usage patterns across two CUDA versions (11.2 and 12.2.2) in two widely used CUDA asynchronous functions, `cudaLaunchKernel` and `cudaMemcpyAsync`, which play fundamental roles in DL model inference. These findings are relevant to future research in ML-enabled real-time systems.

Third, we discuss mitigation strategies for the various problems uncovered herein across different foundational software components. By reducing contention and offering more real-time-compatible locking protocols, we demonstrate their trade-offs and benefits through experiments. Based on prior work in the real-time systems community, we also discuss directions toward a more general solution to these problems (such a solution would require a significant engineering effort that is beyond the scope of this paper). As part of this discussion, we elucidate several actionable items in real-time and ML application development.

[5] A *black-box* software component may or may not have a closed-source implementation. When a component is strictly treated as a black box, the API it provides is considered in development and testing, but its internal implementation, even if available, is not.

[6] We note that glibc 2.23 is relatively dated. However, Ubuntu 16.04 LTS, which is still maintained and supported by Canonical until 2026 [54], uses glibc 2.23 as its standard LIBC library.

Organization. After next providing relevant background (Sec. 2), we present our discoveries (Sec. 3), discuss mitigations (Sec. 4) and future directions (Sec. 5), and conclude (Sec. 6).

2 Background

In this section, we discuss four black boxes used to develop AV software: Linux, QNX, the POSIX-Thread (pthread) library, and CUDA. We focus on aspects of these components that are relevant to our investigation, including their usage in AVs, and provide an overview of relevant real-time concepts.

2.1 Synchronization Primitives in Linux and QNX

Linux and QNX are the most commonly used OSes in AVs due to CUDA's compatibility with them[7] [28]. QNX is a safety-certified, POSIX-compliant real-time OS (RTOS) widely used to expedite the safety-certification process for AV production [5]. However, Linux-based systems are still common in AV development due to Linux's large community of users (offering a large support network) and extensive library of software. To provide certain RTOS features to the Linux kernel, the `PREEMPT_RT` kernel patch is often applied. In this paper, all references to Linux implicitly assume a `PREEMPT_RT`-patched kernel.

The need for process synchronization is fundamental in concurrent programming, and suspension-based locks are commonly used for this purpose. Such locks require involvement with the OS kernel's scheduler to suspend a process. To encapsulate low-level kernel invocations and improve portability, the pthread library provides a standard user-space interface for locking by including functions such as `pthread_mutex` [40, pg. 3,639] and `pthread_rwlock` [40, pg. 3,644]. Henceforth, we refer these two functions as `mutex` and `rwlock` for short. The pthread interface is packaged with the C standard library, LIBC, which comes with QNX and Linux (*i.e.*, glibc) distributions.

In this section, we primarily focus on the implementation of pthread locks in Linux, with a detailed discussion of their implementation in both user space (the glibc pthread library[8]) and kernel space. For QNX, implementation details are not publicly accessible, so we only cover the user-space properties of their pthread lock interface, as described in available documentation [41]. These interfaces will be among the focuses of our investigation.

Futexes. In glibc's pthread implementation on Linux, most suspension-based locks are implemented via the *fast user-space mutex (futex)* [23]. At its core, a futex is a queue-like data structure and synchronization primitive defined in the Linux kernel that allows user-space processes to suspend in a wait-queue until a certain condition is met (*e.g.*, an unlock occurs). In most cases, locking is implemented in user space with two key futex operations: `FUTEX_WAIT` and `FUTEX_WAKE`, as shown in Fig. 1. A futex also provides operations that take process priority into account by using an `rt_mutex` [47] for synchronization. `rt_mutex` satisfies lock requests in priority order and uses priority inheritance. In glibc on Linux, all suspension-based locks are implemented using a fast-path/slow-path approach: if a lock is free, a lock request is handled entirely in user-space (the fast path), otherwise more expensive futex system calls are performed (the slow path).

[7] Other CUDA-supported OSes are Windows and OS X.
[8] There exist several other C standard libraries, such as *musl LIBC*, but this paper only considers glibc and QNX LIBC as they are the standard system C libraries in Linux and QNX.

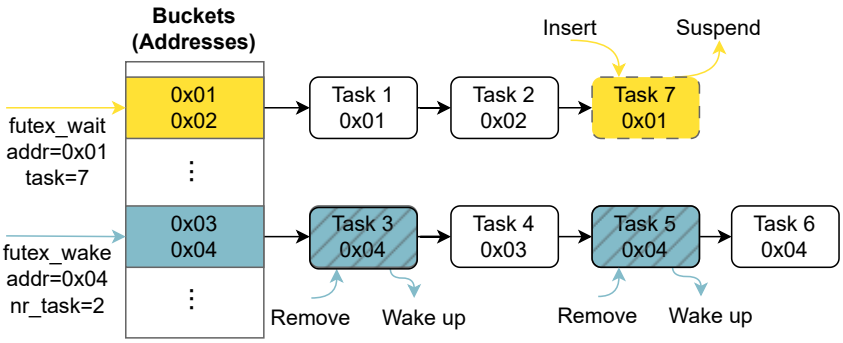


Figure 1 Internal wait-queue structure of a futex in the Linux kernel. An array of hash buckets is used, where each bucket points to a linked-list structure. Two operations are shown: `FUTEX_WAIT` (in yellow) called by Task 7 on address `0x01`; and `FUTEX_WAKE` (in blue) called by an arbitrary task on address `0x04` with two tasks to be woken up (*e.g.*, Task 6 remains suspended).

Mutex locks. The pthread library includes an interface for mutex locks ("mutexes" for short), which provide mutually exclusive resource access. It offers the option of using priority inheritance or immediate priority ceilings[9] ("inheritance" and "ceilings," respectively, for short) [3, 24, 44, 49, 50]. In Linux, glibc implements a pthread mutex with a single futex, with inheritance/ceilings disabled by default. In contrast, in QNX's pthread implementation, inheritance is enforced by default.

Read-write (RW) locks. A RW lock allows concurrent resource access for multiple reader processes, while requiring exclusive access for writer processes. Some RW lock implementations always favor read (resp., write) access requests if any are waiting – these are called *read-preference* (resp., *write-preference*) locks. Such an implementation can starve non-favored requests (illustrated in Appx. A), which is highly problematic in real-time systems. A better alternative that avoids starvation is a *phase-fair* lock [11, 13], which alternates between "read phases" (read requests are satisfied) and "write phases" (a write request is satisfied). The glibc implementation for Linux allows users to choose between a read- and write-preference lock, but defaults to read-preference. QNX instead implements a phase-fair RW lock [42].

2.2 NVIDIA CUDA Framework

NVIDIA is the current market leader in the GPU industry. To enable their GPUs to be applied generally and not just for graphics, NVIDIA created CUDA, a software framework for writing general-purpose parallel programs in C++, among other languages. This parallel computational model provided ML researchers with the computational power necessary for realizing modern DL algorithms. Alternatives to NVIDIA's CUDA do exist, such as AMD's ROCm, but this paper mainly focuses on CUDA-based ML applications, which represent a significant share of modern AV systems.

We later provide an overview of AV systems that discusses the usage of CUDA. Here, we provide a description of CUDA's basic functionality.

[9] In POSIX, this is referred to as `PRIO_PROTECT` [40, pg. 1687]. This mechanism defines a priority ceiling for a resource (higher or equal to the highest-priority task requesting the resource) and raises a lock-holder's priority to the lock priority ceiling upon lock acquisition. This mechanism is also known as *immediate priority inheritance* and the *highest-locker protocol*.

Algorithm 1 Simplified CUDA program.

```
1: function Main
2:     stream := cudaStreamCreate()                    ▷ Initialize CUDA stream
3:     cudaMemcpyAsync(stream, hostIn, deviceIn)       ▷ Copy input from CPU to GPU
4:     cudaLaunchKernel(stream, Kernel)                ▷ Launch the kernel function on GPU
5:     cudaMemcpyAsync(stream, deviceOut, hostOut)     ▷ Copy result from GPU to CPU
6:     cudaStreamSynchronize(stream)                   ▷ Block CPU until the stream is done
7: function Kernel
8:     ...                                             ▷ User-defined GPU computations (not shown)
```

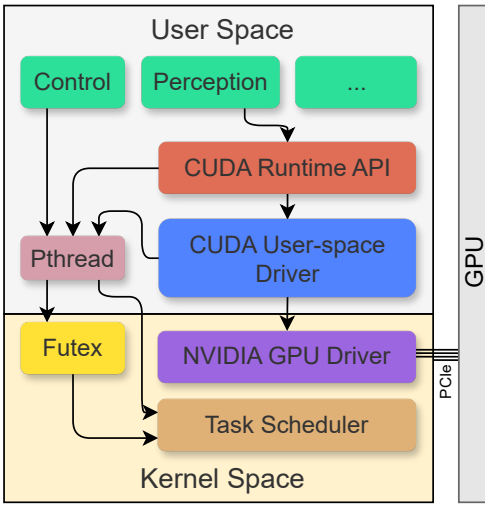


Figure 2 CUDA architecture on Linux.

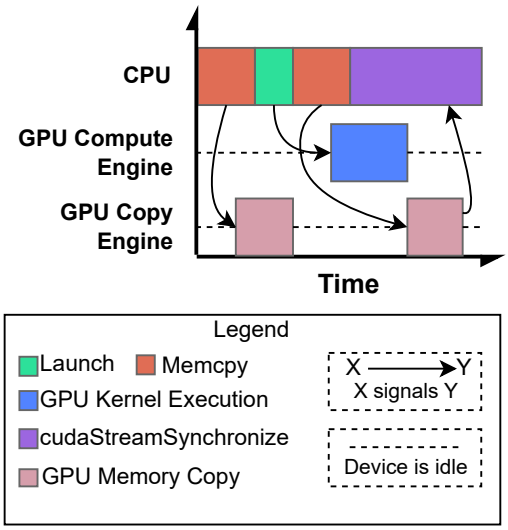


Figure 3 Alg. 1 executed on CPU and GPU.

CUDA architecture overview. Fig. 2 illustrates the overall architecture of CUDA on Linux. To facilitate CPU/GPU communication, NVIDIA distributes a kernel driver for Linux (as well as other OSes, such as QNX). Along with it, NVIDIA provides the CUDA user-space driver, an API that user-space applications can use to interact with a GPU. This driver contains functions for copying data between CPU DRAM and GPU DRAM, functions to send and execute user-written parallelized programs – called *CUDA kernels* ("kernels" for short) – on the GPU, as well as GPU memory management and other GPU utility functions. However, the CUDA user-space driver is cumbersome to use, as it requires tedious setup that involves significant boilerplate code and careful management of GPU resources throughout the lifetime of an application. To enable easier GPU programming, NVIDIA provides the CUDA runtime API, which wraps the unwieldy CUDA user-space driver functions and boilerplate code with a simplified interface. It is for this reason that the CUDA runtime API is by far the most common way developers use CUDA. Surrounding the CUDA interface is an arsenal of DL acceleration libraries and tools allowing for efficient DL training and inference, such as cuDNN, cuBLAS, and TensorRT.

Asynchronous CUDA functions. Two of the most commonly used functions in the CUDA runtime API are `cudaLaunchKernel`, which launches a CUDA kernel to run on a GPU, and `cudaMemcpyAsync`, which initiates a data transfer within, to, or from a GPU. For short, we refer to these two functions as `Launch` and `Memcpy`. Both functions are asynchronous:

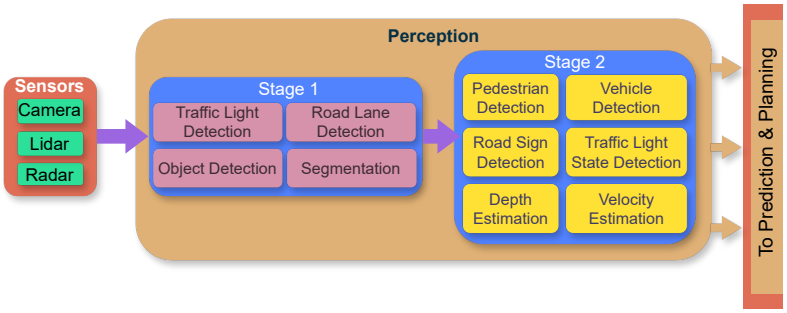


Figure 4 A generic AV workload. Each module (represented by boxes) communicate with each other via data packets. Data dependencies are represented by arrows.

the CPU-side program can immediately proceed to the next instruction after calling these functions, without the need for blocking. This functionality enables the application to execute on both the GPU and CPU in parallel and synchronize (by calling `cudaStreamSynchronize`) only when necessary. Alg. 1 gives an example CUDA program and Fig. 3 shows how this program is executed on both the CPU and GPU.

`Launch` and `Memcpy` will be the focus of our investigation. From our measurements, these functions take about $\leq 30\mu s$ and $1,038\mu s$ (with 1M FP32 data) on the CPU, respectively, when executing in complete isolation. However, we show later that this expected execution time can be greatly exceeded.

2.3 Autonomous-Driving Systems

We now explain how the just-described black boxes (Linux, QNX, the pthread library, and CUDA) are typically used by AV developers.

ML software and hardware in AVs. Modern AVs rely on ML algorithms to achieve high-accuracy perception and prediction, which require GPU acceleration for timely processing. An AV's perception and control systems are composed of several software modules executing in a data-driven manner; each module is activated when input data is provided and passes its results to the next module after completion. For example, a vehicle's perception system may include an obstacle detection module that takes image frames from a camera and point clouds from LiDAR and executes an ML algorithm to detect static obstacles in the frame, as depicted in Fig. 4. Such modules are implemented as processes on an OS. In the case of ML workloads, CUDA is the standard approach for GPU acceleration, as mentioned previously.

On top of CUDA, DL inference is performed with DL runtime libraries. A prominent example is NVIDIA TensorRT, which is widely adopted by ML developers in industry, including AV developers such as ZOOX [35]. TensorRT has built-in support for CUDA and allows for optimizations of DL inference latency on NVIDIA GPUs. With TensorRT, users may compile their ML models into TensorRT inference engine files, which can then be imported into their CUDA programs to perform DL inference in similar ways to CUDA kernel functions, as demonstrated in Sec. 2.2.[10]

[10] Throughout this paper, DL inferences are all conducted with TensorRT, but our results are not limited to this software.

The only automotive embedded context in which NVIDIA supports CUDA and TensorRT is their DRIVE AGX platform, centered around their Tegra SoC (a multicore ARM CPU and an integrated, CUDA-compatible GPU). NVIDIA offers customized Linux and QNX kernels for the DRIVE AGX platform. As NVIDIA only provides first-party support for Linux and QNX on their embedded devices, this shoehorns developers into using these OSes.

OSes in AVs. In AV systems, the use of locks is inherent due to the presence of large amounts of shared data across ML-enabled modules, and to facilitate CPU-GPU synchronization operations. AV systems based on Linux and QNX either directly or indirectly (*e.g.*, via dependency libraries and frameworks like CUDA) use pthread locking interfaces to perform synchronization operations, with their implementations relying heavily on the OS. In addition, the scheduling of tens to hundreds of GPU-enabled processes falls to the OS.

To support developers using their Tegra SoC for embedded AV applications, NVIDIA provides a production-grade AV software package called *DRIVE OS*, which consists of a customized QNX and Linux kernel, foundational libraries, such as LIBC, and an SDK for developing hardware-accelerated AV applications. DRIVE OS serves as a convenient platform to rapidly prototype embedded AV applications, utilizing the black boxes we have discussed.

Safety guarantees. Guaranteeing safety, however, remains a challenge in AV systems. In an AV, inaccurate or delayed execution of critical decision-making software modules can lead to catastrophic outcomes. Thus, for AVs to ever reach mass production, their onboard software must conform to rigid safety requirements. Taken together, the aforementioned black boxes can be used to realize an AV's perception and control software, as demonstrated with DRIVE OS. However, simply joining these black boxes to form an AV platform does not necessarily guarantee safety. For safety requirements to be truly met, the black boxes themselves must provide real-time guarantees.

2.4 Real-Time Systems

A major focus in real-time-systems research is to devise algorithms for verifying system *schedulability*, *i.e.*, that computations complete "on time," which is crucial for AV systems, as noted above. In this section, we provide a brief overview of relevant real-time concepts.

Task scheduling. In the real-time-systems literature, schedulability is studied relative to a *task model*. (For our purposes, the details of defining such a model and differences among different models are unimportant.) The term *task* refers to a sequential program, and a *job* is an instance (*i.e.*, invocation) of a task. Each job requires a set of compute resources (CPU, GPU, *etc.*) in order to execute, and often accesses data resource(s) as well. In the context of an AV system, an ML model is a good example of a task, and each inference performed on the model is a job of the task.[11] A scheduler arbitrates the allocation of compute resources to jobs, whereas a locking protocol arbitrates shared-data accesses. Two types of CPU scheduling algorithms exist, *clock-driven* and *priority-driven*, and both are used in AV systems. Due to the rigidity of clock-driven scheduling, priority-driven scheduling is often preferred in the AV industry. The system considered in our investigation uses priority-driven scheduling, but some of our findings may extend to clock-driven scheduling as well. We assume that m CPU cores exist and that a clustered CPU scheduling algorithm is used where the m cores

[11] This statement is reflective of the assumption that an ML model inference is executed in a single GPU-using process scheduled by the OS. In more complex configurations, such execution could be split across several processes.

are divided into equal-sized clusters of c cores each; each task is assigned to a cluster and globally scheduled on the cores in that cluster. Note that partitioned ($c = 1$) and global ($c = m$) scheduling are special cases.[12]

Pi-blocking. Under the assumption of completely independent jobs, a priority-driven scheduler simply prioritizes jobs by their assigned priorities. However, due to resource contention (*i.e.*, locking), a job may be blocked even if its priority is high enough to be scheduled – this is called a *priority inversion*. More formally, a job suffers *priority-inversion blocking (pi-blocking)* iff it is pending (*i.e.*, not completed) but unscheduled, and there are fewer than c higher-priority pending jobs in its cluster.[13] In a safety-critical AV system, extensive pi-blocking adds to the overhead of the system and can undermine the system's schedulability, so accounting for and minimizing pi-blocking is important for system safety.

Progress mechanisms. In order to bound pi-blocking, a real-time locking protocol must ensure lock-holder execution using a suitable *progress mechanism* [13] so that pi-blocked jobs "make progress" towards lock acquisition. Various progress mechanisms have been considered in work on real-time multiprocessor systems, including priority inheritance, priority donation, priority boosting, and migratory inheritance [10]. Some progress mechanisms may only properly function under certain scheduling assumptions. For example, priority inheritance is only effective under global scheduling ($c = m$), and only if the right locking protocol is used [12], but not under purely clustered (or partitioned) scheduling ($c < m$) [7, page 125]. Thus, these mechanisms must be integrated into the OS alongside its scheduler to effectively manage pi-blocking. Prior work on mutexes has shown that $O(m)$ per-request pi-blocking, which is asymptotically optimal, can be achieved under a variety of schedulers (partitioned, clustered, global) [1,6,8,9,14,55]. Similarly, for RW locks, asymptotically optimal per-request pi-blocking of $O(m)$ for writers and $O(1)$ for readers is possible [11, 13].

Hereafter, we use the terms "process" and "task" somewhat interchangeably, but do favor the former (resp., latter) when discussing systems-oriented (resp., analysis-oriented) issues.

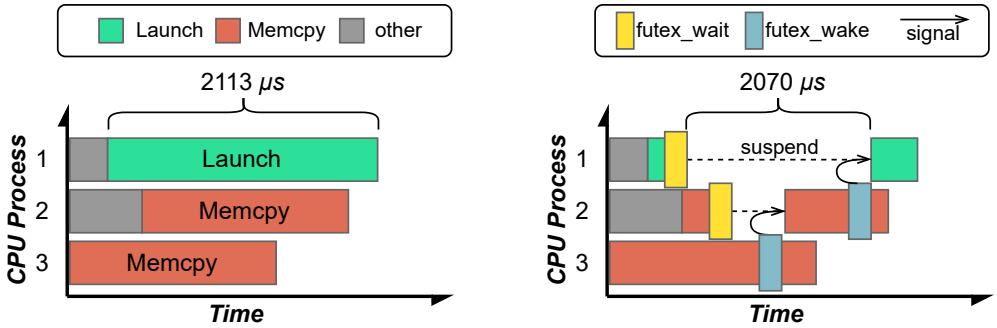
3 Investigation and Discoveries

Our investigation was motivated by CUDA-related delays seen in an actual AV. In Sec. 3.1 below, we elaborate on these delays and describe the methods we used to find their source. Then, in Sec. 3.2, we report on broader findings concerning Linux, QNX, glibc, and CUDA, obtained using these same methods, and discuss their implications for safety-critical systems.

3.1 Case Study: Motivation, Methods, and Root Cause

The AV system we considered is under commercial development at WeRide. The vehicle testing platform is an x86 system with an NVIDIA Ampere-generation GPU. The machine was configured with Linux 5.40, NVIDIA driver 460.84, CUDA 11.2, and glibc 2.23.[14] The vehicle's AV software stack runs in a data-driven manner, similar to the architecture schema described in Sec. 2.

[12] In practice, processor affinity masks could be used so that scheduling is not purely cluster-based, but considering affinities would be a major distraction in defining scheduling and locking terminology.
[13] Throughout this paper, we assume the use of *suspension-oblivious* analysis [11], wherein suspension time is analytically treated as computation time when checking schedulability.
[14] This version setup was used to reflect the setup of the AV platform used when the issue introduced in this section was discovered. It does not reflect the current setup in said AV system's production.



(a) The abnormal latency observed in CUDA kernel launch, revealed by Nsight Systems tracing.

(b) System tracing with KUtrace reveals the source of latency as suspension caused by futex operations.

Figure 5 Tracing results of CUDA asynchronous functions.

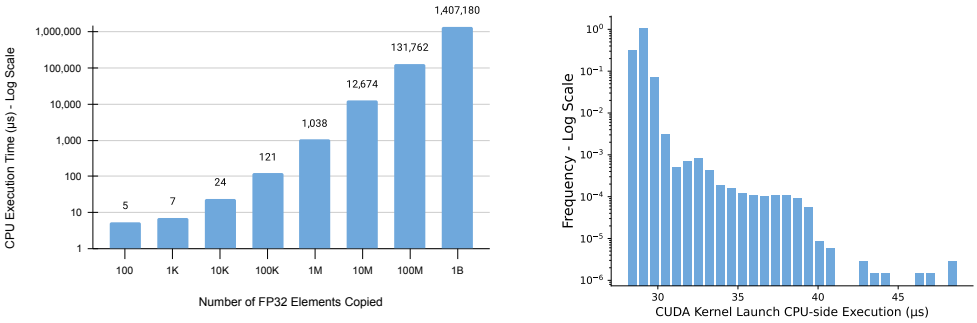
While testing the vehicle in 2022, the system's safety monitor recorded a series of latency spikes (30% increase in system end-to-end latency compared to other instances). The latency spikes would manifest as modules with abnormally long execution times, ultimately causing the system to revert to its backup system as it was unable to react to real-world events on time. This section describes the steps we followed to investigate these latency spikes.

Investigation overview. Our investigation involved the following steps. First, we reproduced the anomalous timing behavior and identified the modules involved in the observed high latency. Module profiling further revealed two CUDA functions as the main latency source. We subsequently employed system tracing and *shimming* (detailed later) techniques to pinpoint pthread RW lock starvation as the main culprit. Analyzing the glibc pthread source code ultimately revealed a critical flaw in its RW lock implementation, which proved to be the underlying cause of the high latency. We now discuss our methodology in detail.

Reproduction. By utilizing our proprietary in-house simulation and profiling tools[15] on hardware and software identical to the AV, we were able to reproduce latency spikes similar to that observed on the AV after continuously running the AV software for three hours. Preliminary inspections of the AV profile revealed that all affected modules were performing ML inference using CUDA when latency spikes occurred. We initially suspected GPU over-utilization as the culprit. However, our hardware system monitor revealed that the GPU compute engine was idle at the time of delays. Thus, we instead focused our attention on the CPU-side behavior during ML inference.

Profiling CUDA. We further profiled the offending modules with NVIDIA Nsight Systems [33], a GPU and CUDA profiling tool, in the context of the aforementioned reproduction in AV simulation. Nsight Systems provided statistics and visualization of the exact timing of GPU execution and CUDA functions as the models executed. This approach confirmed that the latency source was not caused by on-GPU contention but rather by CPU-side CUDA functions launching GPU operations. Fig. 5a illustrates these scenarios. As depicted, the

[15] Our AV simulation and profiling software are not publicly accessible. However, alternatives can be devised from open-source resources such as the CARLA simulator [17], LTTng [30], and KUtrace [51].



(a) Measured `Memcpy` CPU-side execution time when copying various amounts of FP32 data to the GPU. For reference, an RGB 1920×1080 (FHD) image requires roughly 6M FP32 elements.

(b) Distribution in histogram of 1M samples of `Launch` CPU-side execution time. This data demonstrates that most kernel launches can finish on the CPU-side within 30 μs.

Figure 6 Measurements of execution time for GPU memory copy and kernel launch operations without any interference.

CUDA function `Launch`, when called by modules from the CPU, incurred significant delays of up to 2113 μs, whereas the function would normally take less than 30 μs when running in complete isolation, as shown in Fig. 6b.

The almost concurrent completion of a kernel launch and `Memcpy` observed in our trace (Fig. 5a) implicates on-CPU interference between these two CUDA functions as a potential cause. Even though these functions utilize two disjoint GPU engines (*i.e.*, compute and copy engine) and may run independently on the GPU [39],[16] it was plausible that they were contending for shared data structures on the CPU. However, the CUDA runtime is a closed-source black box, making code analysis infeasible. Therefore, to identify the internal operations causing the delays in CUDA functions, we had to devise alternative techniques.

System tracing. System tracing captures low-level system behavior, such as suspensions, syscalls, I/O accesses, and faults, so it is useful in holistically understanding intra- and inter-process interactions in black-box software over a period of time. Such an approach can provide clues to infer the intent behind a black box's operations.

In our investigation, we used KUtrace [52] [51], a low-overhead system tracing tool. Tracing a system involves running the KUtrace control program, which generates a trace timeline of events in the OS. Fig. 7 shows the structure of tracing timelines produced by KUtrace in the form of interactive HTML webpages (Appendix B provides an actual output).

Having already identified the two CUDA asynchronous functions, `Launch` and `Memcpy`, as the latency source, we relied on synthetic tracing experiments instead of simulating the entire AV system, as this provides a more simplified, isolated environment for trace analysis. Alg. 2 outlines the program we used to perform tracing experiments without involving the AV software.

By studying CUDA-related system behavior as captured with KUtrace, we discovered that CUDA memory-copy functions and kernel launches synchronize via several futexes, causing prolonged suspension in the `FUTEX_WAIT` system call, as shown in Fig. 5b. These suspensions

[16] In this paper, we avoid discussing the intricacies of these two operations on the GPU for the purpose of simplicity and to focus purely on their CPU behaviors.

Algorithm 2 Tracing experiment program.

```
1: function MemcpyThread
2:     for ever do cudaMemcpyAsync(...)          ▷ Repeatedly call CUDA memcpy function
3: function KernelThread
4:     for ever do cudaLaunchKernel(...)         ▷ Repeatedly launch CUDA kernel
5: function Main
6:     for each tid in 1...m do CreateThread(MemcpyThread)   ▷ Start multiple memcpy threads
7:     CreateThread(KernelThread)                ▷ Start one kernel launch thread
```

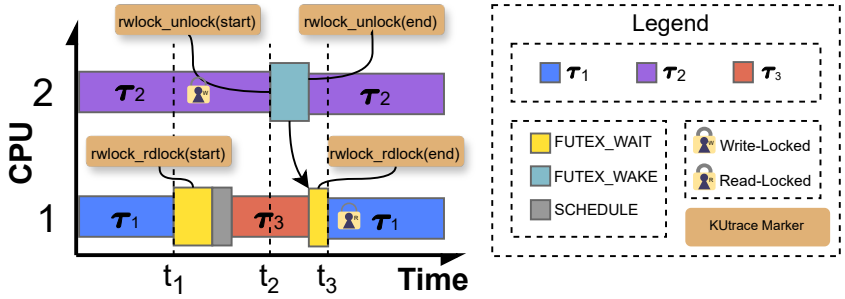


Figure 7 Illustration of a KUtrace output trace for three tasks running on two CPU cores. Task τ_1 has higher priority than τ_2, which has higher priority than τ_3. This figure depicts RW lock contention between τ_1 (reader) and τ_2 (writer). Narrow bands are executions in user space, and thick bands are those in kernel space.

within CUDA kernel launches were the main source of the studied latency. However, to better understand what triggered the system calls, more information was needed concerning the CUDA functions' internal behavior.

Tracing locking behavior in CUDA. As discussed in Sec. 2, rarely do user libraries such as CUDA invoke futex operations directly. Instead, futexes are used to implement abstracted locks in libraries such as the pthread library. Thus, we sought to determine if the CUDA functions use such abstraction layers atop futex, and, if so, which specific functions are used.

Although CUDA's reliance on the pthread library is undocumented, confirming this fact takes little effort. The CUDA runtime API is implemented in the `libcudart.so` shared-object file, and the CUDA user-space driver is in `libcuda.so`. By using the `ldd` command in Linux, which lists dynamic dependencies of shared-object files, we found that both link to `libpthread.so`, implemented in glibc on Linux by default.

However, identifying the specific pthread locking functions used by CUDA proved difficult, as source code is required to trigger trace events before and after calling the pthread functions in CUDA. To overcome this challenge, we utilized a technique known as *shimming*. In this technique, a custom shim version of an existing function overrides the original function at runtime, without modifying the original application. This can be achieved by setting the `LD_PRELOAD` environment variable to the shared-object file of the custom shim functions.

In our case, we applied shimming to emit KUtrace event markers around the CUDA functions' calls to pthread locking functions, most notably its mutex and RW lock functions. Fig. 8 illustrates this technique. We accomplished this by first creating shim versions of the pthread locking functions (*e.g.*, `pthread_mutex_lock`) in a shared-object file, with KUtrace events signaling the beginning and completion of the pthread functions. The shim functions then return back to the calling CUDA functions. The resulting trace from executing our program (Alg. 2), now containing tracing events for pthread locks, confirmed CUDA's usage of pthread mutex and RW locks and revealed their exact usage pattern.

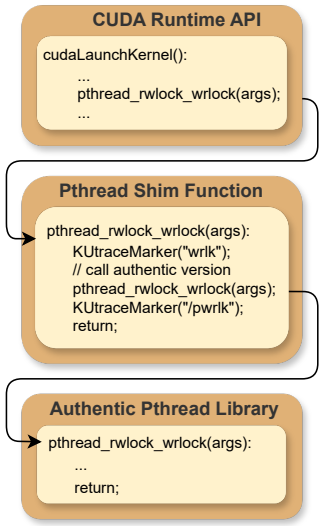


Figure 8 Shimming flowchart. The shim function can be compiled and loaded via `LD_PRELOAD`.

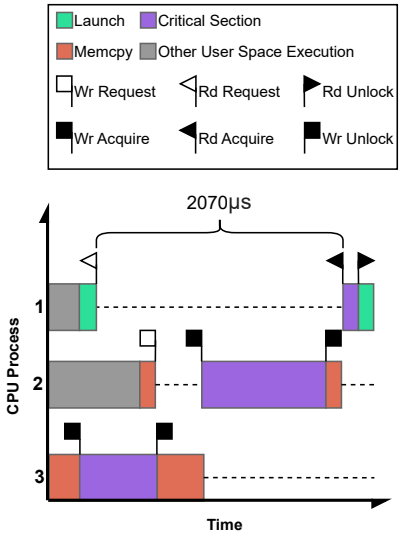


Figure 9 Shimming reveals pthread RW lock triggered mutex operation and caused suspension.

pthread locking behavior in CUDA. Our trace data showed that CUDA's asynchronous function calls use a fixed set of two RW locks and seven mutexes to launch a GPU operation, employing a complex and nested locking pattern composed of several critical sections. We discuss the locking patterns more precisely in Sec. 3.2; here, we focus on the main cause of the latency in our motivating scenario.

Both `Launch` and `Memcpy` require read and write accesses, respectively, to a shared RW lock. As demonstrated in Fig. 9, the pthread RW lock read requests triggered the `FUTEX_WAIT` system call and directly caused the prolonged suspension, ultimately causing the aforementioned failure in our AV system. As this trace data matches what we would have seen in a write-preference pthread RW lock, demonstrated in Fig. 14a, it may be the case that CUDA uses this policy intentionally.

However, by adding additional CUDA kernel launch threads in our program (Alg. 2), our data revealed that the pthread RW locking performed in CUDA functions does not follow a strict read-preference or write-preference policy. Instead, it follows a read-preference policy during read phases, but a write-preference policy during write phases (illustration can be found in Appx. A). In other words, if the lock was unlocked by a writer (write phase), pending write requests would be satisfied first. However, if the lock was held by a reader (read phase), newly issued read requests would be satisfied even if there were pending write requests. This violates the definition for both read- and write-preference RW locks.

To further understand the RW lock's internal logic, we shimmed the RW lock initialization function (`rwlock_init`), where its policy is declared. This method revealed that CUDA was in fact configuring the lock to use the default policy (*i.e.*, read-preference), despite our observations clearly displaying violations of that policy. Therefore, a careful analysis of the pthread RW lock implementation was warranted in Linux glibc.

Identifying the root cause. We consequently examined the glibc source code [25, 27] that arbitrates the next-to-be-satisfied lock request in the pthread RW lock. Alg. 3 shows a pseudo-code for glibc's (version 2.23) implementation of `rwlock_unlock`. We discovered a critical flaw within this function: if the lock is set to the read-preference policy, then

Algorithm 3 `pthread_rwlock_unlock`.

```
1: function RWLOCK-UNLOCK(rwlock)
2:     if rwlock.is-reader then rwlock.curr-num-readers−−          ▷ First, release the lock
3:     else rwlock.writer := NULL
4:     if rwlock.curr-num-readers = 0 then      ▷ If this thread is a writer or the last active reader
5:         if rwlock.waiting-writers then                          ▷ If there's a pending write request
6:             FutexWake(rwlock.wr-futex, 1)          ▷ Then give the lock to the next pending writer
7:         else
8:             FutexWake(rwlock.rd-futex, MAX_INT)    ▷ Otherwise, give the lock to all pending readers
```

after a writer unlocks, instead of favoring pending read requests as expected, the lock will unconditionally satisfy a pending write request, if one exists (Alg. 3, lines 7–8). Existing in glibc 2.4 through 2.24, this implementation can starve read and write requests. Although a read-preference lock causing write-request starvation is permissible, causing read-request starvation is an error. This is the exact scenario we observed in our trace Fig. 9, where read requests in kernel launches are starved by write requests in the `Memcpy` functions.

Implications in AV and real-time systems. We identified the root cause of our AV system failure. However, our findings posed additional questions regarding the extent to which the RW lock bug and design deficiencies affect DL systems in AVs. As the two considered CUDA functions play a significant role in DL inference (detailed in Sec. 3.2), having lock starvation is gravely dangerous in the context of AV systems, as such delays are undocumented and remain unknown to AV developers. Furthermore, analysis in prior work on GPU-enabled real-time systems, such as [2, 21], fails to consider the potentially indefinite synchronization between these two functions, which are regarded as independent on the GPU. We thus conducted additional experiments to test how the two CUDA functions' synchronization patterns affect DL inference across a wider range of system configurations.

3.2 Broader Findings

In this section, we discuss additional findings on the CPU-side synchronization behavior of the two CUDA functions `Launch` and `Memcpy` in the context of DL inference. To this end, we demonstrate the extensive usage of these two functions during DL inference, detail the locking patterns used in these CUDA functions, and reveal our findings regarding CUDA, glibc (Linux), and QNX Libc using techniques similar to those discussed in Sec. 3.1. All our findings in this section are based on experiments of discrete GPU architectures using NVIDIA driver 550.54.14, CUDA 12.2.2, glibc 2.38, and QNX 8.0 unless otherwise indicated.[17]

CUDA functions in DL inference. CUDA operations are used extensively to accelerate DL inference tasks. Although we cannot use our proprietary DL model architectures to demonstrate this effect, publicly accessible DL models and AV workloads can serve as meaningful surrogates. Tbl. 1 illustrates common DL models used for computer-vision tasks and the number of CUDA function calls made while performing inference with these models.[18] An AV workload consists of several of these models; for example, six such perception-related DL modules are identified in [32]. Additional, more specialized DL modules are also present in

[17] These software versions are the most recent as of writing.
[18] The number of CUDA function calls may vary depending on the specific model variant, ML compiler, and GPU architecture; the statistics provided here only serve as a representation.

Table 1 CUDA function invocations in common perception DL models and their execution times, measured on an NVIDIA Titan V in a single run.

DL model	`Launch`		`Memcpy`		GPU time
	count	avg. latency	count	avg. latency	
ViT-S [19] [38]	60	5.5 μs	2	187 μs	2,208 μs
RegNet-Y [43] [37]	47	6.0 μs	2	1,793 μs	5,275 μs
DeiT-B [53] [36]	60	5.8 μs	2	1,700 μs	4,902 μs
DETR-R50 [16] [31]	180	5.9 μs	3	743 μs	7,045 μs
SegFormer-B1 [56] [34]	88	6.2 μs	2	6,747 μs	16,222 μs

AVs aside from the six named in [32], such as obstacle detection and traffic-light classification. In some cases, separate DL pipelines exist for processing LiDAR and camera data in the AV system.[19] Based on this information and the data from Tbl. 1, an AV system containing ten DL components could (as a fairly conservative estimate) contain 470 `Launch` calls and 20 `Memcpy` calls in a single run. As an AV system may perform over ten runs per second to react to real-world events, even intermittent delays in these two functions could prove disastrous.

Pthread locking usage in CUDA functions. By using the shimming technique mentioned in Sec. 3.1, we reveal the exact locking operations performed and their timings in `Launch` `Memcpy` using both CUDA 11.2 and 12.2.2 (the most recent version at the time of writing) in Fig. 10. Our data further demonstrates the extensive usage of locking invoked by CUDA functions in an AV workload, implying that tens of thousands of such invocations can take place in a single run. Combining our previous finding that the RW lock operations may introduce indefinite delays, the data clearly demonstrates the pervasive but latent presence of locking operations in AV systems, and their high potential for causing AV failures, especially as the AV's backup system is not immune to such delays either.[20]

We observed in our experiments that CUDA's locking behavior is highly inconsistent across versions. Fig. 10 reveals the detailed locking usage of locking within CUDA kernel launch and memory copy functions across three variations. Fig. 10 (top) shows the locking pattern used in CUDA 11.2 with NVIDIA driver 460, on which our investigation in Sec. 3.1 is based. Using NVIDIA GPU driver version 525 and CUDA 12.2.2 (Fig. 10, middle), we discovered that kernel launches are no longer using any RW lock. However, using the most updated driver version 550, CUDA 12.2.2 (Fig. 10, bottom) uses additional locking operations in its kernel launches.

Another more baffling locking usage can be observed when using glibc 2.23 with CUDA 11.2 and driver 460. With these versions, before invoking `rwlock_rdlock` (resp., `_wrlock`), CUDA will first attempt to call `rwlock_timedwrlock` (resp., `_timedrdlock`) with invalid time as arguments on the same lock address (not shown in Fig. 10 for simplicity). We hypothesize that CUDA is attempting to exploit the "fast path" for the RW lock, since even with invalid time arguments, the glibc pthread implementation will still attempt to acquire the lock first before failing [26]. However, this does not explain why write (resp., read) locks are attempted before applying for read (resp., write) locks.

[19] This paper does not take into account the practice of DL multi-tasking, which allows models to perform multiple tasks in a shared DL backbone (e.g., RegNet).

[20] To assist researchers in understanding CUDA locking usage in future releases, we have provided a source code for this purpose: `https://github.com/sizheliu-unc/ECRTS24/tree/main/cuda_lock_stats`.

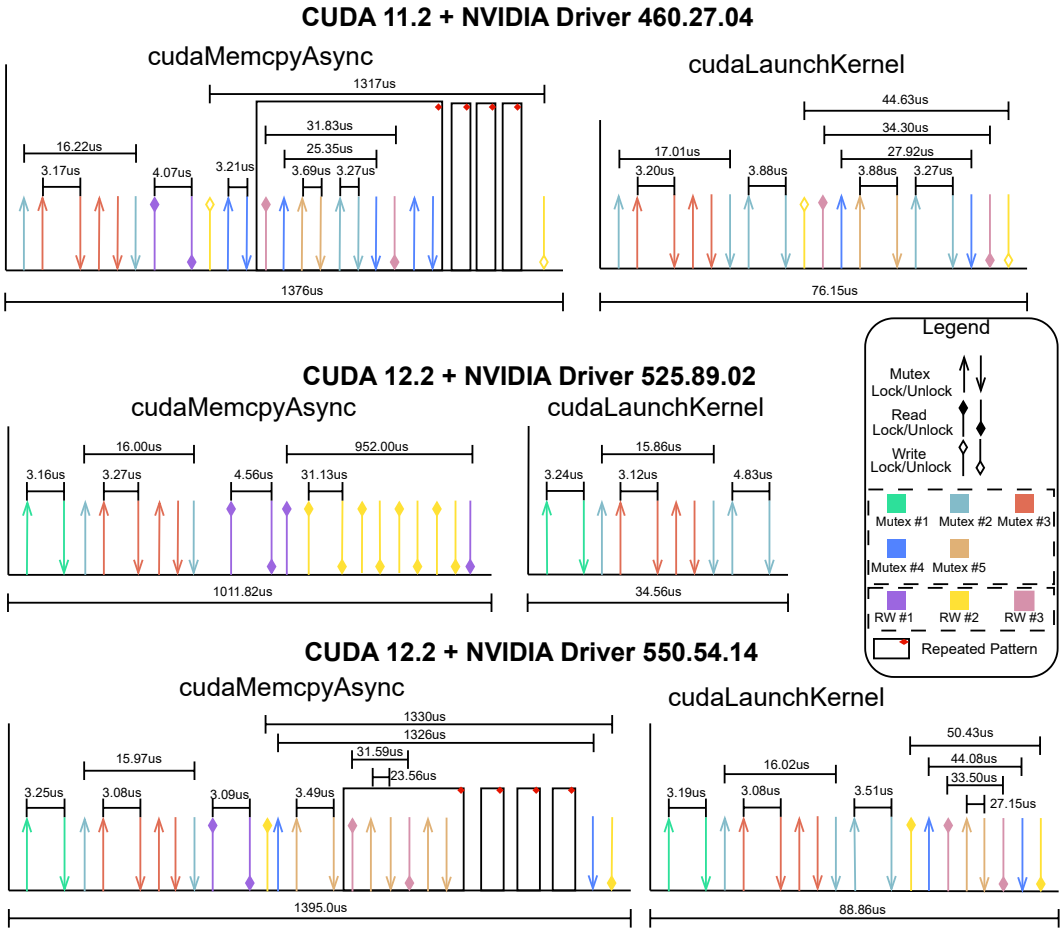


Figure 10 The RW lock and mutex usage in functions `Memcpy` (1M FP32 data copy) and `Launch` across versions. Colored arrows denote a lock and unlock pair. Locks with the same number but not within the same driver version may not represent the same functionality. Notice that **RW lock #2** (dark yellow) is used in both functions in driver version 460, whereas **mutex #4** (light blue) is used in both functions in driver version 550. Only locks shared between the two functions are shown. Timings of repeated locking patterns are not shown.

Linux CUDA locking is contention-prone. As shown in Fig. 10 (top), CUDA 11.2 shares the same RW lock between `Launch` and `Memcpy`. This locking pattern subjects CUDA functions to prolonged delays when executing them concurrently. Even though these RW locks (Fig. 10, top, RW lock #2) are dropped in CUDA 12.2 with driver 525, driver 550 (Fig. 10, bottom) instead replaces them with a mutex (#4) in effect. Therefore, scenarios similar to Fig. 9 can still occur even with the most updated software (CUDA 12.2, driver 550, glibc 2.38), with the only difference being that the problematic RW lock in 11.2 is replaced by a mutex. Due to its long critical section in `Memcpy` (1326μs with 1M FP32 data), having concurrent executions of `Memcpy` and `Launch` can have a cascading effect. As an extreme example, when launching 100 threads of `Memcpy` and `Launch` together, we are able to induce 110 ms delay to both functions. Such extensive blocking is clearly unacceptable for time-sensitive applications such as AVs.

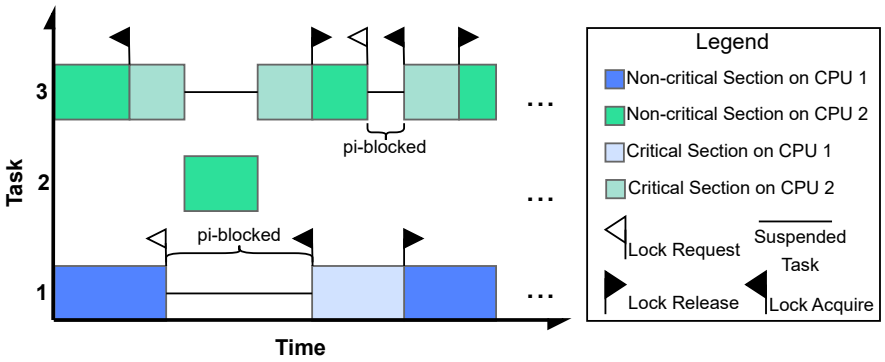


Figure 11 A trace showing priority inversion in CUDA, in which priority is ordered by index. Task τ_1 is running `Launch` and is pinned to CPU 1, task τ_2 is an arbitrary task on CPU 2, and task τ_3 is running `Memcpy` copy on CPU 2. In an AV context, the high-priority task might execute a pedestrian-detection algorithm, and the low-priority task might be a data transfer for turn-signal detection. This situation could lead to unacceptable delays in a critical AV function.

glibc RW lock is starvation-prone, causing CUDA delay. Also problematic is the glibc RW lock implementation, even with version 2.38. As we discussed, preference RW locks, the only available policies in glibc, will cause starvation to either readers or writers. Moreover, the glibc RW lock implementation is not equipped with any progress mechanism. As explained in Sec. 2, a progress mechanism guarantees lock-holder execution. Without such a mechanism, a CUDA 11.2 kernel launch can be preempted and cause pi-blocking while holding a RW lock. This problem is present in all glibc versions. Fig. 11 demonstrates this effect. By increasing the number of non-GPU-using tasks (depicted as τ_2 in Fig. 11), we induced indefinite delay to `Launch` in CUDA 11.2. This would prevent applications from performing any ML inference.

Linux CUDA's mutex is starvation-prone. Even though glibc mutexes do offer inheritance as a progress mechanism, it is not used in CUDA versions 11.2 and 12.2.2. As shown in Fig. 10, both CUDA versions internally utilize mutexes as another synchronization mechanism between kernel launches. Using the shimming method, we were able to confirm that on Linux, the mutexes used by CUDA (both versions) do not have inheritance enabled, again allowing for potentially indefinite delays similar to Fig. 11. We also observed that inheritance is sometimes, but not always, used on the NVIDIA Jetson Orin integrated GPU, indicating some awareness of pi-blocking among CUDA developers.

Partial mitigations are offered in QNX LIBC. On QNX, RW locks are implemented with a phase-fair mechanism (discussed in Sec. 2). However, through experiments, we found QNX's RW lock, similar to that of glibc, is not equipped with any progress mechanism, and thus is prone to starvation. Consequently, CUDA applications on DRIVE OS for QNX can still experience indefinite delays as shown in Fig. 11. QNX is being actively targeted for AV production (not just testing), and the existence of such a potentially disastrous issue in this safety-certified RTOS demonstrates the danger of a black-box design approach. Even if such a black box has passed some level of certification and is widely adopted, it still must undergo a thorough inspection for the specific safety-critical context to which it is applied.

On the other hand, the inheritance mutex is used by default on QNX, so inheritance is enabled for all CUDA mutexes. However, inheritance is only effective under specific locking protocols designed for global scheduling [7, page 125], whereas in most AV systems, clustered

scheduling is used. The ineffectiveness of inheritance for a partitioned workload stems from the fact that priorities have relative meaning only within a partition. This partial mitigation may still induce high overheads to an AV's complex workload under high contention.

Developers don't understand the limits of priority inheritance on multicore. Our investigation into the vehicle's latency spikes required us to read the documentation or source code for the pthread, Linux, QNX, and glibc black boxes. Through the course of our investigation, it became apparent that there is a major disconnect with respect to the term "priority inversion" between the real-time-systems community and the developers who make these black boxes. We realized many OS developers tend to believe that a task suffers pi-blocking only when it is blocked by a lower-priority task. However, this definition is only accurate on a uniprocessor [7]. Fundamentally, pi-blocking occurs when a task should be scheduled but cannot due to blocking. The formal definition of pi-blocking, which is of relevance to multicore machines, can be found in Sec. 2.4.

This pi-blocking disconnect has led many developers and users of these black boxes to believe that inheritance [50] and ceilings [3, 24, 44, 49] are panaceas to control pi-blocking. As a result, they are the only progress mechanisms provided for synchronization mechanisms on POSIX-compliant systems, including Linux and QNX. However, these mechanisms break down under non-global scheduling [7], which is common in embedded AV systems.

In addition, many user-space and kernel-space locking mechanisms on Linux and QNX are based on priority ordering, believed by many to be a crucial concept in minimizing worst-case pi-blocking, even though it has been shown to be less effective than FIFO ordering for this purpose [11]. With n tasks running on m CPU cores, in the worst case, even with global scheduling, the inheritance mutex induces $\Omega(n)$ pi-blocking [12, 20]. Furthermore, the FMLP [6], which the ceiling-based mutex in glibc highly resembles,[21] induces $\Theta(n)$ pi-blocking [6, 12]. Thus, both are far from the optimal pi-blocking of $O(m)$ mentioned in Sec. 2.4. These non-optimal pi-blocking bounds are unsatisfactory for systems like AVs that likely involve heavy resource contention and where the number of tasks is far greater than the number of cores.

Unfortunately, only these two progress mechanisms, inheritance and ceilings, exist for mutexes in POSIX, Linux, QNX, and glibc. Additionally, across all these black-box components, no progress mechanism is adopted at all for RW locks, rendering this common synchronization primitive unsafe in real-time applications such as AVs.

4 Mitigations

In this section, we discuss three specific strategies for mitigating the problems covered in Sec. 3. These mitigation strategies, even when combined, are by no means a full solution for ameliorating the shortcomings of the black boxes we have covered. Rather, we intend for them to serve just as a starting point from which more exhaustive solutions can be obtained through future work.

[21] Because glibc's implementation results in all blocked tasks having the same priority in futex, blocked tasks are subject to FIFO ordering, resembling the FMLP.

Algorithm 4 CUDA graph usage.

```
1: function Main
2:     cudaStreamBeginCapture(stream, graph)                ▷ Begin to capture graph
3:       cudaMemcpyAsync(stream, hostIn, deviceIn)          ▷ Launch operations on the captured stream
4:       cudaLaunchKernel(stream, Kernel)
5:       cudaMemcpyAsync(stream, deviceOut, hostOut)
6:     cudaStreamEndCapture(stream, graph)                  ▷ End graph capture
7:     cudaGraphLaunch(graph)                               ▷ Actual inference, launch the entire graph
8:     cudaStreamSynchronize(stream)
9:     ...                                                  ▷ The graph object can be used for consequent launches (not shown)
```

Table 2 Comparison of end-to-end execution time, launch time, and number of locking operations of DL inference between using CUDA graphs and not in a single run. Measurements were collected with 20 seconds of back-to-back execution. Launch time was measured as the difference in time between the first GPU work launch (including memcpy) and the beginning of the first GPU work.

DL model	Avg. End-to-end time		Avg. Launch time		# of Locking	
	W/ Graph	W/o Graph	W/	W/o	W/	W/o
ViT-S	$2,356\,\mu s$	$2,378\,\mu s$	$144\,\mu s$	$152\,\mu s$	147	1,088
RegNet-Y	$5,464\,\mu s$	$5,491\,\mu s$	$148\,\mu s$	$155\,\mu s$	171	1,242
DeiT-B	$5,052\,\mu s$	$5,073\,\mu s$	$153\,\mu s$	$154\,\mu s$	327	1,412
DETR	$7,846\,\mu s$	$8,432\,\mu s$	$153\,\mu s$	$154\,\mu s$	192	3,566
SegFormer	$16,385\,\mu s$	$16,743\,\mu s$	$269\,\mu s$	$294\,\mu s$	71	1,543

4.1 Reducing Contention Using CUDA Graphs

In Sec. 3, we recounted our findings that CPU-side lock contention can significantly delay CUDA functions. It would follow that reducing the number of CUDA operations in the system would reduce lock contention, and hence reduce the likelihood of latency spikes. One option to achieve this is to use *CUDA Graphs* [18] to combine multiple CUDA kernel and memory-copy functions into one, reducing the number of launch operations in the system. As demonstrated in Alg. 4, modifying existing CUDA programs to use CUDA Graphs involves adding only a few lines of code. Lines 2-6 specify the graph and only need to execute once. The resulting *graph* object can then be used multiple times. More complex dependency relations can also be specified using *CUDA events*. In addition, NVIDIA's ML acceleration tools, such as TensorRT, natively support CUDA Graphs.

After using CUDA graphs in our AV system modules, the latency spikes from CUDA operation launches were no longer observed in our testing. It would appear using CUDA Graph solved the problem completely. As demonstrated in Tbl. 2, CUDA Graph can mildly reduce the DL model's execution time and launch time. This is achieved by reducing the communication cost between CPU and GPU (detailed in Appx. C).

In addition, CUDA Graph may significantly reduce locking usage. As shown in Tbl. 2, the graph launch of RegNet-Y contains 171 mutex and RW lock operations, whereas 1,242 locking operations are used when not using CUDA Graph. However, this does not completely eliminate contentions. From our experiments, more than one hundred mutex operations are involved even for launching only four GPU operations (three memory copies and one kernel). The number of locking is proportional to the number of GPU operations and the size of the memory copy. Our experiments show that each 1M (resp. 1K) FP32 `cudaMemcpyAsync` corresponds to ~ 30 (resp. ~ 10) mutex operations in the CUDA Graph launch. This number does not scale linearly with respect to the number of `cudaLaunchKernel`: with CUDA Graph,

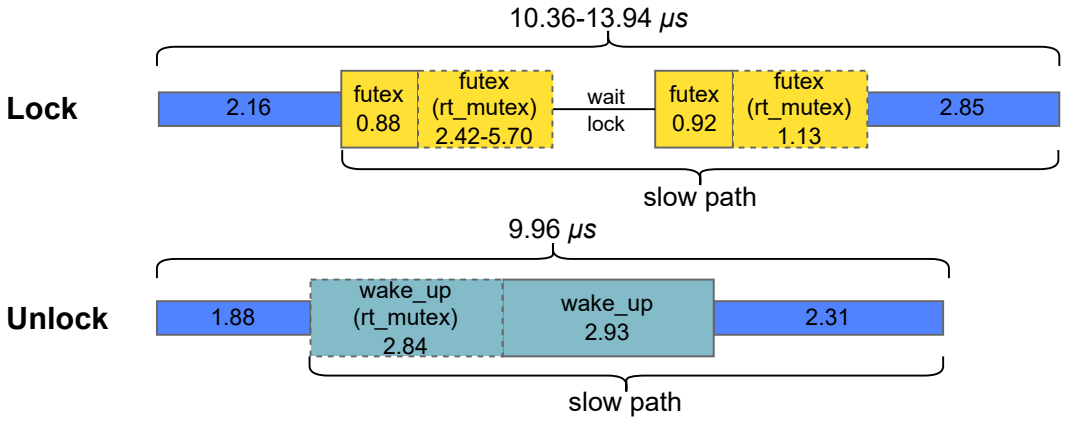


Figure 12 Detailed timing of mutex lock and unlock operation. Numbers represents the execution time of each operations in μs. Thick bands represents system calls (i.e. kernel space executions), and thin bands represents user space executions. "**Slow path**" is present only under lock contention. `rt_mutex` is present only when using the `PRIO_INHERIT` policy and may vary in execution time depending on the number of nested locks, whether lock holder's priority should be adjusted, etc..

launching DeiT-B (60 `Launch`) contains 327 locking, whereas launching DETR-R50 (180 `Launch`) involves 192 locking. Since no write locks are involved in launching a CUDA Graph, RW-lock contention (as in Fig. 9) between graph launches is eliminated.

One caveat of using CUDA graph is that certain operations cannot be expressed with CUDA Graph. For example, conditional statements acting as a control switch are not supported for CUDA 12.2 and before. Such statements are needed, for instance, if the output of one ML model inference dictates which ML model inference is performed next. Such CUDA operations must therefore be broken into separate graphs. A complex AV system often involves control switches, and with many graphs executing in parallel, lock contention may still occur. In our case, CUDA graph is effective in reducing contention and is adopted in our AV platform.

4.2 Enforcing Inheritance Mutexes

As discussed in Sec. 3, even though priority inheritance only provides a weak solution to reduce pi-blocking, it is the only existing viable progress mechanism natively supported by Linux and QNX. The pthread mutex is the only lock type utilizing this progress mechanism. However, on Linux systems, the inheritance option must be specified during mutex initialization. As we discovered, on x86 Linux, the most common development platform for ML, CUDA does not use inheritance mutexes. To enforce inheritance-mutex usage in CUDA or any other software modules, a simple solution is to use the same shimming technique described in Sec. 3.

We applied this technique to our AV system and found that it increased the overall maximum latency of our AV modules (with CUDA 12.2 and glibc 2.38) by 5%. We hypothesize the reason for this increase is likely due to the overhead caused by its underlying `rt_mutex`, which involves additional steps to modify task priorities compared to regular futex operations. As demonstrated by the mutex data in Fig. 12, the execution time of lock and unlock operations (excluding suspension) is significantly higher for the inheritance mutex compared to the default policy. Recall from our discussion in Sec. 3 that thousands of such lock operations take place in an AV during a single run. This additional overhead, when multiplied by the thousands, can have a significant impact on the system.

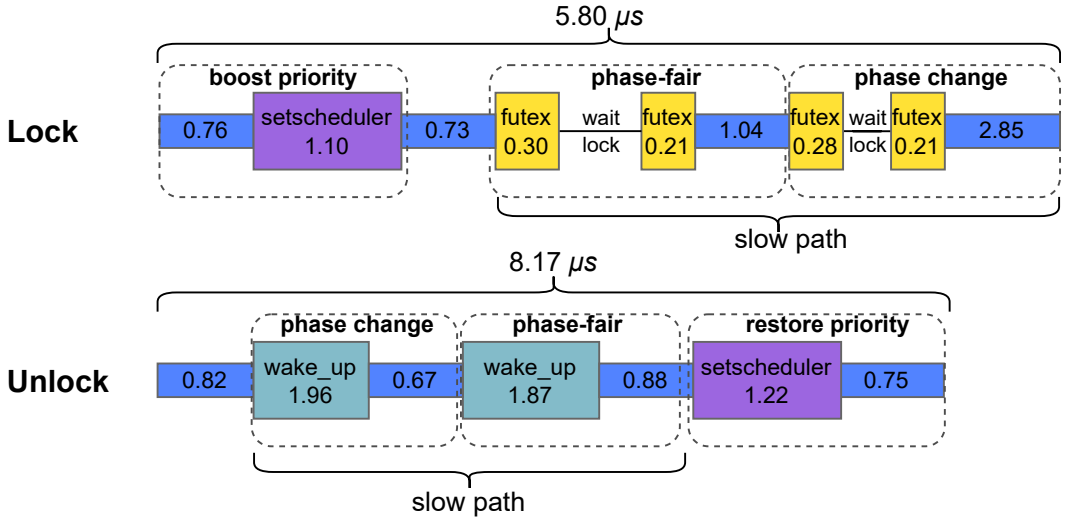


Figure 13 Detailed timing of RW lock and unlock operation. Blocks in dashed lines represents related functionalities. "**Boost/restore priority**" block is present only in our RW-lock priority boosting progress mechanism. "**Phase-fair**" block is present iff (1) using phase-fair or write-preference, when applying for a read lock, the lock is in a read phase, and pending write requests exist; (2) when applying for a write lock, other pending writers exists. "**Phase change**" occurs when the lock's phase (read/write) switches.

Moreover, the inheritance mutex is not an optimal solution, as mentioned in Sec. 3; its $\Omega(n)$ worst-case pi-blocking can induce high costs in complex systems like AVs. This result demonstrates a current dilemma faced by the AV industry: either use partial solutions like inheritance mutexes for safety but with high costs, or use non-real-time variants that provide no guarantee but offer better performance.

4.3 Implementing Phase-Fair RW Locks in Glibc

To reduce preference-induced RW-lock starvation, one strategy is to use a phase-fair RW lock instead. As mentioned in Sec. 2, a phase-fair RW lock is already integrated in the QNX pthread library, but glibc does not provide this option. By modifying the pthread RW lock implementation in glibc,[22] we were able to enforce this mechanism on all RW lock usage, thereby avoiding both write- or read-request starvation under high contention.

This only partially addresses RW-lock-related problems on Linux and QNX, however. As explained in Sec. 3, the lack of suitable progress mechanisms is an additional challenge. One of the known progress mechanisms suitable for phase-fair RW locks is *priority donation* [14], which enables optimal pi-blocking under clustered (and hence partitioned and global) scheduling. This mechanism requires knowledge of all tasks in the system instead of only lock-requesting ones, so implementing it would involve radical modifications to Linux's scheduler, breaking portability to other OSes, like QNX. We instead opted to implement a more basic progress mechanism similar to the ceiling mutex mentioned in Sec. 2, where the lock-requesting process's priority is boosted upon blocking and lock acquisition. This locking mechanism, as shown in Fig. 13, still adds significant overhead to the lock compared to

[22] Our implementation of phase-fair RW lock with priority boosting can be found at `https://github.com/sizheliu-unc/ECRTS24/blob/main/glibc-2.38-phase-fair-boosting.patch`.

the default (read-preference) RW lock but can prevent indefinite pi-blocking and starvation (depicted in Fig. 11) from occurring in the system. Due to the overhead involved in priority boosting, we are unable to integrate this approach into our system without increasing the overall latency in our particular AV system.

5 Future Directions

We have discussed how lock contention and problematic lock implementations can lead to system failures on an actual AV. In this section, we shift our focus to more broadly discussing action items based on our findings from Sec. 3.

For Linux and QNX developers. Suitable locking protocols and progress mechanisms are fundamental for safety-critical real-time systems such as AVs. As noted in [15], migratory priority inheritance could be a suitable candidate to limit pi-blocking in Linux. The comparative study in [57] also showed that the FMLP outperforms inheritance mutexes (referred to as the PIP in [57]) in practical circumstances. Adopting migratory priority inheritance in Linux would be ideal, but it would entail some significant changes to the futex, `rt_mutex`, and scheduler implementations. Implementing the FMLP, on the other hand, should be a good starting point for Linux developers, as it only affects the data structure behind `rt_mutex`. As for QNX, since its main target is vehicle production, the lack of a suitable progress mechanism for locking and especially RW locks must be urgently addressed.

Another factor that Linux and QNX developers need to consider is that these foundational software components are not regularly updated when used in systems such as AVs. In this case, "upgrading the software" is both a risky and costly solution, as the new upgrades may break dependencies and require rebuilding the entire system. Therefore, bug fixes to existing issues should be applied to both the new updates and the affected existing versions.

For CUDA developers. CUDA is closed-source, so the precise reasons for why it uses locking so extensively remain as a hidden detail. A possible way forward would be for CUDA to provide options for real-time systems to use inheritance mutexes, and avoid using RW locks in the implementation due to the lack of a progress mechanism for such locks on both Linux and QNX. Furthermore, CUDA's documentation should provide more detail on the synchronization between CUDA functions so that developers are aware of possible delays.

For AV developers. As we have demonstrated, timing-related issues encountered in AV systems may be deeply rooted inside several black-box components. AV developers must realize that high reliability is not equivalent to guaranteed behavior. A full inspection of these black boxes is optimal but expensive and sometimes infeasible. Case-by-case analysis thus becomes the only means for AV developers to understand abnormal system behaviors. This approach is insufficient to truly guarantee safety, however. Thus, a joint effort involving developers working on AV systems, CUDA, and relevant OSes is needed to introduce robust real-time guarantees into these components.

For real-time researchers. As the locks used by CUDA do not provide any real-time guarantees, the common practice of having separate GPU engine locks, as proposed in [21], will thus still be prone to delays; instead, a single real-time-compatible mutex guarding these CUDA functions is required, but this can be inefficient in practical situations. Moreover, the implementation of futex on Linux also implies that some GPU tasks may be inadvertently

ordered by priority due to CPU-side lock contention instead of the expected FIFO ordering, potentially causing unexpected delays. A plausible solution for real-time researchers is to use the shimming method to replace CUDA's locking usages with real-time-compatible variants.

6 Conclusions

Software developers have always taken advantage of existing code bases to increase productivity, so it is unsurprising that AV developers have adopted the same approach. However, AVs fall within a unique category of applications where any system misbehavior can have catastrophic consequences. Thus, it is simply not acceptable to take a "black-box" approach to AV system design without a thorough understanding of the components being applied. In this paper, we have peered into certain aspects of several such black boxes – Linux, QNX, the glibc pthreads library, and CUDA – and have shown that they can be a source of consequential timing-related problems. We have also proposed partial mitigations for these problems. Unfortunately, a full mitigation would require a complete re-design to incorporate real-time principles into AV-related components, and this would be a herculean task. At the very least, we hope this paper brings some awareness of the dangers of failing to "look inside the box."

In future work, we intend to examine other black boxes commonly used in the AV industry. Chief among them is ROS (the Robot Operating System) [46], which is widely being used to facilitate modular system development in AVs. ROS (even recent "real-time" variants of it) was not designed with real-time predictability as a first-class concern, so its use in contexts where real-time safety is paramount is highly questionable.

References

1 S. Ahmed and J. Anderson. Optimal multiprocessor locking protocols under fifo scheduling. In *Proceedings of the 35th Euromicro Conference on Real-Time Systems*, pages 16.1–16.21, July 2023.

2 T. Amert, Z. Tong, S. Voronov, J. Bakita, F.D. Smith, and J. Anderson. TimeWall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 455–468, December 2021.

3 N. C. Audsley, A. Burns, and A. J. Wellings. Deadline monotonic scheduling theory and application. *Control Engineering Practice*, 1(1):71–78, 1993. doi:10.1016/0967-0661(93)92105-D.

4 M. R. Bachute and J. M. Subhedar. Autonomous driving architectures: Insights of machine learning and deep learning algorithms. *Machine Learning with Applications*, 6:100164, 2021. doi:10.1016/j.mlwa.2021.100164.

5 BlackBerry QNX Safety Certifications, Compliance and Conformance. https://blackberry.qnx.com/en/developers/certifications. Accessed: 2024-05-10.

6 A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80. IEEE, August 2007.

7 B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

8 B. Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 292–302, July 2013.

9 B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, pages 61–71, July 2014.

10 B. Brandenburg. Multiprocessor real-time locking protocols: A systematic review. *CoRR*, abs/1909.09600, 2019.

11 B. Brandenburg and J. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 184–193, July 2009.

12 B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60. IEEE Press, December 2010.

13 B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the ACM International Conference on Embedded Software*, pages 69–78. ACM, October 2011.

14 B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342, 2014.

15 B. Brandenburg and A. Bastoni. The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors. In *14th Real-Time Linux Workshop*, pages 67–86. Real-Time Linux Foundation, 2012.

16 N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-end object detection with transformers. In A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, editors, *Computer Vision – ECCV 2020*, pages 213–229, Cham, 2020. Springer International Publishing.

17 Carla Simulator. `https://carla.org/`. Accessed: 2024-05-10.

18 Getting Started with CUDA Graphs. `https://developer.nvidia.com/blog/cuda-graphs/`. Accessed: 2024-05-10.

19 A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL: `https://openreview.net/forum?id=YicbFdNTTy`.

20 A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 377–386. IEEE, December 2009.

21 G. Elliott. *Real-Time Scheduling of GPUs, with Applications in Advanced Automotive Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2015.

22 G. Elliott and J. Anderson. The limitations of fixed-priority interrupt handling in preempt rt and alternative approaches. In *Proceedings of the 14th OSADL Real-Time Linux Workshop*, pages 149–155, 2012. URL: `https://api.semanticscholar.org/CorpusID:1503941`.

23 H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *AUUG Conference Proceedings*, volume 85, pages 479–495. AUUG, Inc, 2002.

24 B. Gallmeister and C. Lanier. Early experience with POSIX 1003.4 and POSIX 1003.4A. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 190–198. IEEE, December 1991.

25 Bootlin Elixir cross referencer – GLIBC 2.23. `https://elixir.bootlin.com/glibc/glibc-2.23/source`. Accessed: 2024-05-10.

26 Bootlin Elixir cross referencer – GLIBC 2.23 – pthread_rwlock_timedrdlock. `https://elixir.bootlin.com/glibc/glibc-2.23/source/nptl/pthread_rwlock_timedrdlock.c`. Accessed: 2024-05-10.

27 The GNU C library (GLIBC). `https://elixir.bootlin.com/glibc/glibc-2.38/source/sysdeps/nptl`. Accessed: 2024-05-10.

28 L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2021. `doi:10.1109/JIOT.2020.3043716`.

29 Sizhe Liu. Source code for ECRTS 2024 paper Autonomy Today: Many Delay-Prone Black Boxes. Software, swhId: swh:1:dir:825b348c28207aef045b6565ae0e977add42d44c, (visited on 06/06/2024). URL: https://github.com/sizheliu-unc/ECRTS24.

30 LTTng site. https://lttng.org/. Accessed: 2024-05-10.

31 Meta Research – DETR. https://github.com/facebookresearch/detr. Accessed: 2024-05-10.

32 K. Muhammad, A. Ullah, J. Lloret, J. D. Ser, and V. H. C. de Albuquerque. Deep learning for safe autonomous driving: Current challenges and future directions. *IEEE Transactions on Intelligent Transportation Systems*, 22(7):4316–4336, 2021. doi:10.1109/TITS.2020.3032227.

33 NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: 2024-05-10.

34 NVIDIA Research Projects – SegFormer. https://github.com/NVlabs/SegFormer. Accessed: 2024-05-10.

35 NVIDIA TensorRT official website. https://developer.nvidia.com/tensorrt. Accessed: 2024-05-10.

36 ONNX model – DeiT-B . https://github.com/onnx/models/blob/main/Computer_Vision/deit3_base_patch16_224_Opset17_timm/deit3_base_patch16_224_Opset17.onnx. Accessed: 2024-05-10.

37 ONNX model – RegNet-Y. https://github.com/onnx/models/blob/main/Computer_Vision/regnet_y_16gf_Opset16_torch_hub/regnet_y_16gf_Opset16.onnx. Accessed: 2024-05-10.

38 ONNX model – ViT-S. https://github.com/onnx/models/blob/main/Computer_Vision/vit_small_patch16_224_Opset16_timm/vit_small_patch16_224_Opset16.onnx. Accessed: 2024-05-10.

39 How to Overlap Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/. Accessed: 2024-05-10.

40 IEEE standard for information technology–portable operating system interface (POSIX(TM)) base specifications, issue 7, 2018. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), pp. 1–3951. doi:10.1109/IEEESTD.2018.8277153.

41 QNX Neutrino Realtime Operating System: Library Reference. http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_lib_ref%2Fp%2Fpthread_rwlock_rdlock.html. Accessed: 2024-05-10.

42 QNX Reader/writer locks. https://www.qnx.com/developers/docs/7.1/index.html#com.qnx.doc.neutrino.sys_arch/topic/kernel_Reader_writer_locks.html. Accessed: 2024-05-10.

43 I. Radosavovic, R. Kosaraju, R. Girshick, K. He, and P. Dollar. Designing network design spaces. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10425–10433, Los Alamitos, CA, USA, June 2020. IEEE Computer Society. doi:10.1109/CVPR42600.2020.01044.

44 R. Rajkumar, L. Sha, and J. P. Lehoczky. An experimental investigation of synchronization protocols. *IEEE Real-Time Systems Newsletter*, 5(2-3):11–17, 1989.

45 F. Reghenzani, G. Massari, and W. Fornaciari. The real-time Linux kernel: A survey on PREEMPT_RT. *ACM Comput. Surv.*, 52(1), February 2019. doi:10.1145/3297714.

46 ROS: Home. https://www.ros.org/. Accessed: 2024-05-10.

47 RT-mutex subsystem with PI support. https://docs.kernel.org/locking/rt-mutex.html. Accessed: 2024-05-10.

48 C. Scordino and G. Lipari. Linux and real-time: Current approaches and future opportunities. In *IEEE International Congress ANIPLA*, 2006.

49 L. Sha and J. B. Goodenough. Real-time scheduling theory and Ada. *Computer*, 23(4):53–62, 1990. doi:10.1109/2.55469.

50 L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. doi:10.1109/12.57058.

51 R. L. Sites. Benchmarking "Hello, world!": Six different views of the execution of "Hello, world!" show what is often missing in today's tools. *ACM Queue*, 16(5):54–80, October 2018. doi:10.1145/3291276.3291278.

52 R. L. Sites. *Understanding software dynamics*. Addison-Wesley Professional Computing Series. Addison Wesley, Boston, MA, February 2022.

53 H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jegou. Training data-efficient image transformers &; distillation through attention. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10347–10357. PMLR, 18–24 July 2021. URL: https://proceedings.mlr.press/v139/touvron21a.html.

54 Ubuntu 16.04 LTS (Xenial Xerus). https://ubuntu.com/16-04. Accessed: 2024-025-10.

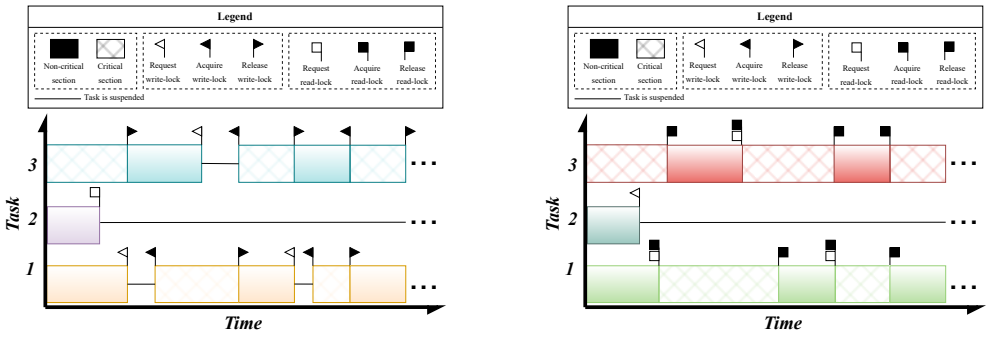
55 B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 223–232, July 2012.

56 E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo. Segformer: Simple and efficient design for semantic segmentation with transformers. In *Neural Information Processing Systems (NeurIPS)*, 2021.

57 M. Yang, A. Wieder, and B. Brandenburg. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *2015 IEEE Real-Time Systems Symposium*, pages 1–12, 2015. doi:10.1109/RTSS.2015.8.

A Preference RW Lock Starvation

As illustrated in Fig. 14, preference RW locks can induce indefinite delays to read- (in case of write-preference) and write-requests (in case of read-preference).



(a) Illustration of read request starvation in a write-preference RW lock. Task τ_2's read request is starved by periodic write requests from tasks τ_1 and τ_3, causing the lock to never enter the read phase.

(b) Illustration of write request starvation in a read-preference RW lock. Task τ_2's write request is starved by periodic read requests from tasks τ_1 and τ_3, causing the lock to never enter the write phase.

Figure 14 Two types of request starvation with preference RW locks, the only options in glibc.

B KUtrace Output

Fig. 15 and Fig. 16 are screen shots taken directly from KUtrace outputs. Specifically, Fig. 15 represents a trace profile for `Launch` and Fig. 16 shows a trace where two processes calling `Memcpy` contends for the same RW lock. The thick band shown in Fig. 16 represents futex operations ("wakeup" and "futex").

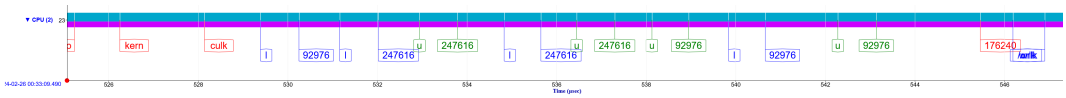


Figure 15 KUtrace snippet capturing a test program (running on CPU #23) at the beginning of a `Launch` CPU-side execution. Here, the label "kern" represents when the test program called the CUDA kernel, and "culk" represents when the shimmed version of `Launch` actually began. The "l" labels represent when mutex lock operations starts, followed by another label listing the lock's address. The "u" labels represent a mutex unlock operation, also followed by a label representing the lock's address.

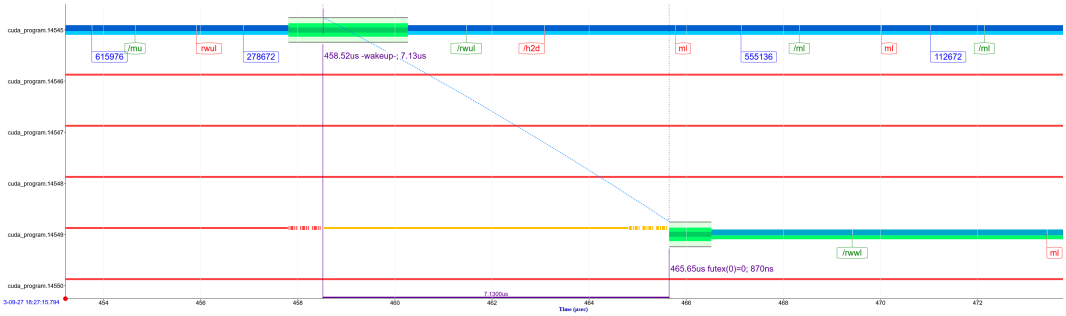


Figure 16 KUtrace snippet capturing a test CUDA program executing simultaneous `Memcpy` calls across many threads. The KUtrace events timeline has been organized by PID (shown on the left), instead of by CPU. The "ml" marker indicates the beginning of a `mutex_lock` call, "mu" indicates the beginning of a `mutex_unlock` call, and "rwul" indicates the beginning of a `rwlock_unlock` call. The "/ml" marker indicates a `mutex_lock` call has returned, "/mu" marker indicates a `mutex_unlock` call has returned, "/rwul" indicates a `rwlock_unlock` call has returned, "/rwwl" indicates a `rwlock_wrlock` call has returned, and "/h2d" represents the end of a `Memcpy` call that copies data from the CPU to GPU. The marker labels containing numbers will follow a lock operation marker, and indicates the address of the lock the operation was passed as an argument. At the top, the thread with PID 14545 starts unlocking ("rwul" marker label) a RW lock with address 278672, just before the 456 μs timestamp. The unlock operation calls the `futex` syscall, notifying any waiters of the newly freed lock. This notifies the thread with PID 14549, just before the 466 timestamp, causing it get the RW lock in writer mode ("rwul" marker label) mode.

C CUDA Graph

Fig. 17 demonstrates how CUDA Graph can reduce the end-to-end latency of the DL model. Without using CUDA Graph, launching a series of small kernels (as most DL models do) can result in idleness in GPU when the instructions are yet to be sent from the CPU (shown as gaps within kernel executions in Fig. 17). CUDA Graph resolves this issue by batching GPU operations together and send them to the GPU at once.

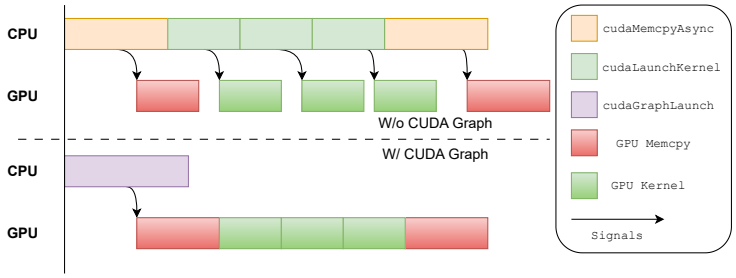


Figure 17 A high-level illustration of GPU operations launched with and without using CUDA Graph. CUDA graph reduces the "gaps" between GPU executions by batching GPU operations.