Want Predictable GPU Execution? Beware SMIs!

Rohan Wagle*, Zelin Tong*, Richard L. Sites†, and James H. Anderson*
*Department of Computer Science
The University of North Carolina at Chapel Hill
†Google Emeritus

***Abstract*—It is common practice today to design complex safety-critical systems by repurposing hardware and software components originally designed for other contexts and using such components in a "black-box" fashion. However, if a black box's inner workings are not fully understood, then this can be unsafe. This paper reports on an investigation pertaining to a black box that is important for autonomous systems, namely NVIDIA's CUDA GPU framework. This investigation was motivated by certain timing glitches in CUDA kernels reported in the literature. After extensive tracing and testing efforts, the culprit causing these glitches was surprisingly found to be not CUDA-related at all, but rather delays due to system management interrupts (SMIs), a known source of timing unpredictability on x86 machines that is rarely if ever mentioned in work on real-time GPU usage. The effects of these SMIs are invisible to the operating system and can cause all cores on an x86 machine to become unavailable for over 20ms! This paper describes the methods used to uncover this timing-glitch source. It also discusses some lessons learned when trying to validate the timing behavior of black-box components.**

I. Introduction

In work on safety-critical embedded systems, the introduction of AI-based autonomous features ranks as a fundamental sea change. The potential impact here can best be seen in the automotive industry, where companies are fiercely competing to field ever more sophisticated autonomous features in their product lines. The hoped for culmination of this competition is full autonomy at mass-market scales. The stakes here are high: the companies (and countries) that get there first will be in a commanding position to influence how autonomy-related capabilities evolve for decades to come. This high-stakes competition has resulted in significant pressure to innovate quickly with respect to key technologies for autonomy, such as perception and decision-making capabilities.

Enabling innovation: the "black-box" approach. This pressure to innovate quickly has led to a "black-box" approach to system design, with off-the-shelf software and hardware components, originally intended for other contexts, repurposed for safety-critical settings. In the automotive domain, two notable examples of repurposed black-box components are ROS (the Robot Operating System) [10], and NVIDIA's CUDA API [25], which facilitates using NVIDIA GPUs to accelerate mathematical computations common of AI applications.

While this black-box approach can speed innovation, it can also lead to unsafe system designs if a black box's inner workings are not fully understood. While this statement may seem self-evident with respect to logical correctness, it is also true of temporal correctness, *i.e.*, that black-box computations predictably complete within specified time limits. Although black-box design approaches certainly did not originate with work on autonomy, the ubiquitous usage of such approaches in work on this topic raises disturbing safety concerns.

Focus of this paper. This paper reports on an investigation undertaken to peer inside CUDA, an important black box for autonomy. This investigation was motivated by rare "timing glitches" observed in prior work by our group [13], wherein certain CUDA functions took several orders of magnitude longer than usual to complete. Such glitches have also been reported by others on online developer forums [7], [11], [12]. We are also aware of one autonomous-driving company (which we cannot name due to privacy concerns) where large GPU execution-time variations were seen in a vehicle under test.

In this paper, we examine the CUDA-related timing glitches reported in [13], and explain various hypotheses we formulated concerning these glitches. We also elucidate the experimental tools and processes we used to confirm or refute each hypothesis. Parts of the CUDA framework are closed-source, so understanding its functioning in depth can be challenging; we explain how we addressed such challenges. We also cover various lessons learned in this effort that may be useful to others wishing to peer inside other important black boxes.

Summary of our findings. Our investigation extends the initial examination of the timing glitches reported in [13]. We initially suspected the CUDA runtime, or one of its dependency libraries, as the culprit. To pinpoint the source of the problem, we performed extensive tracing, using the KUtrace tracing framework [29], of the CUDA application considered in [13]. This provided a clearer picture of the internal operations within the application and the CUDA runtime, but the complexity of the application's code stymied attempts to find simple answers. This led us to seek a minimal example CUDA program that exhibits the latency, while using only a few instructions. The minimal example program eventually revealed that the latency can affect any x86 application, not just CUDA programs (which was a surprise to us). After much effort, we eventually uncovered the true culprit: system management interrupts (SMIs) associated with a hidden motherboard firmware management routine that monitors and

*This work was supported by NSF grants CPS 1837337, CPS 2038855, CPS 2038960, and CNS 2151829, and ONR grant N00014-20-1-2698.

controls fan speed. *This routine is completely invisible to the operating system (OS), and may arbitrarily preempt all CPUs on the system simultaneously, while maintaining complete control over the system for over 20ms*!

The potential for SMIs to disrupt real-time workloads on x86 machines is an issue that has been known for at least two decades [24] and is mentioned in documentation from Red Hat [3] and RT Linux [4]. However, due to the complex and black-box nature of the CUDA framework, it may contain many unknown sources of latency. Thus, SMI-induced latencies may easily be attributed to CUDA mistakenly. This can misdirect debugging efforts from finding true latency sources. Despite the well-documented disruptive nature of SMIs, their potential impact on CUDA seems to be not widely appreciated. Indeed, while a wealth of papers have been written where worst-case GPU kernel execution times are a subject of consideration (an up-to-date citation list, which we omit here due to space constraints, can be found [15]), we are aware of no such papers that mention the impact of SMIs. The primary contribution of this paper lies in documenting this impact, so that others working on real-time GPU-enabled systems will be aware of it. A secondary contribution lies in discussing techniques that can be applied to black-box system components (even closed-source ones) to understand their timing behavior.

Organization. In the rest of this paper, we provide necessary background (Sec. II), discuss the research agenda we followed to discover the source of the timing glitches reported in [13], and provide further detail on this latency source (Sec. III). Then, we discuss the broader implications of our investigation (Sec. IV), explain the necessity of documenting SMIs as a CUDA latency source (Sec. V), and conclude (Sec. VI).

II. Background

In this section, we provide relevant background information.

A brief introduction to CUDA. NVIDIA developed the CUDA platform and API to facilitate GPU acceleration of non-graphics-related computing tasks. The general structure of a CUDA program is as follows: **(i)** allocate necessary memory on the GPU; **(ii)** copy input data from the CPU to the GPU; **(iii)** execute a GPU program called a *kernel*; **(iv)** copy the results from the GPU back to the CPU; **(v)** free unneeded memory. The API functions that implement these operations are provided by NVIDIA in the form of a runtime library. Later in this section, we provide additional details concerning CUDA that are relevant to our investigation.

TimeWall. This paper was motivated by GPU-related timing glitches noticed by our group in work on a framework called TimeWall [13]. Thus, a brief description of TimeWall is in order. TimeWall is a Linux-based (more precisely, LITMUS$^{\mathrm{RT}}$-based [8]) framework for isolating modular software components on a multicore platform augmented with GPUs as computational accelerators. Each component encapsulates a real-time workload. For the sake of understanding our investigation, it is sufficient to know that for TimeWall to ensure temporal correctness, it requires knowledge of the execution times of each GPU computation, including the time required to launch it from the CPU, execute it on the GPU, and then transmit its results back to the CPU.

Device	**GPU**	**CPU**			
Statistic	max	99.9^{th}	99.95^{th}	99.99^{th}	max
Kernel1	27	154	161	1342	1391
Kernel2	42	146	148	163	1388

TABLE I: Kernel execution times (in microseconds) for two different kernels as measured on the GPU (using `nvprof`) and on the CPU (using `clock_gettime()`) [13].

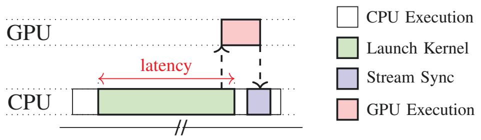
High tail latencies. In [13], a sample CUDA application, *Histogram of Oriented Gradients (HOG)*, was instrumented to work within the TimeWall framework. To determine the application's execution time, a measurement-based approach was used. The considered GPU kernels typically required tens of microseconds to execute on the GPU itself. Naturally, when measuring the full execution time of a GPU kernel on the CPU (which adds the overhead time of launching it and obtaining its results), increased values were seen. At quite high percentiles (99.9^{th} to 99.95^{th}), a reasonable extent of increase was noted, but when moving to worst-case observed values (which may not even be true worst-case values), large increases of almost two orders of magnitude were seen. Tbl. I shows some example data that was reported as part of that investigation that illustrates this increase. Our work in this paper is directed at understanding the reason for the high *tail latencies* in this execution-time distribution data.

Tail-latency sources. [13] describes a cursory investigation into the source of these tail latencies that was performed. It was found that the latencies were attributed to two CPU-bound functions inside the closed-sourced CUDA runtime API library, namely, `cudaLaunchKernel`, which is responsible for initiating CUDA kernels, and `cudaStreamSynchronize`, which awaits kernel completion. Fig. 1 illustrates the scenarios where these functions were seen to incur significant latency.

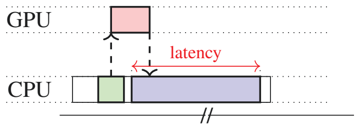
Ruled-out causes. In the initial investigation, several steps were taken to rule out certain possible latency sources. First, interrupt-related causes were eliminated by directing all maskable interrupts to dedicated CPU cores not involved in GPU work. Second, various power-management causes such as CPU frequency scaling and low-powered CPU states (C-states) were ruled out by disabling such features in the BIOS and the OS kernel. Finally, thermal throttling was ruled out as a cause by monitoring core frequencies in the experiments.

Tail-latency mitigation. In TimeWall, high tail lantencies are mitigated via a GPU budget-enforcement mechanism that aborts any computation that runs for too long. While budget enforcement is always advisable for safety, fully understanding why these latencies occur would obviously be desirable.

Further CUDA details. CUDA provides an API for sup-



(a) An anomalous `cudaLaunchKernel` call.



(b) An anomalous `cudaStreamSynchronize` call.

Fig. 1: High-latency scenarios in TimeWall.

porting GPU operations (data copies and kernel launches), known as the *CUDA User-Space Driver API*, referred to here as *CUDA-UDA* for brevity. CUDA-UDA is distributed as a shared-object library file, `libcuda.so`, with the standard NVIDIA GPU driver. User-space applications link to this library file to use CUDA-UDA. However, while CUDA-UDA provides fine-grained GPU control, writing CUDA applications using it directly can be tedious, as this requires setting up and managing several GPU data structures. To alleviate developers from this, NVIDIA provides the *CUDA runtime API*, which wraps CUDA-UDA to automatically handle the tedious GPU set-up and management chores. We henceforth refer to the CUDA runtime API as *CUDA-RT* for brevity. CUDA-RT is distributed as a linkable shared-object library, `libcudart.so`, separate from the NVIDIA GPU driver installation package. As the latency could have been caused by either interface, it was important in our investigation that we study both separately. Given that the latencies were observed in CUDA-RT functions, `cudaLaunchKernel` and `cudaStreamSynchronize`, it was important that we understood how they wrap functions in CUDA-UDA.

These interactions take place within the broader context illustrated in Fig. 2. At the base, the NVIDIA GPU kernel module for Linux facilitates communication between the CPU and GPU. Sitting atop the NVIDIA GPU kernel module is CUDA-UDA. It acts as an interface between user-space applications and the GPU kernel driver. Data and CUDA kernel functions given to CUDA-UDA are relayed to the NVIDIA GPU kernel driver for transmission to the GPU. User-space applications communicate with CUDA-UDA by linking to its shared-object library file, which contains an implementation of CUDA-UDA's functions. Similar to using CUDA-UDA, user-space applications may use CUDA-RT by linking to its shared-object library file, which contains the runtime functions' implementation. As noted above, user-space applications may use CUDA either through CUDA-UDA or CUDA-RT.

KUtrace. In order to obtain an understanding of CUDA's timing behavior, we relied heavily on a powerful tracing framework called KUtrace. KUtrace is a low-overhead software

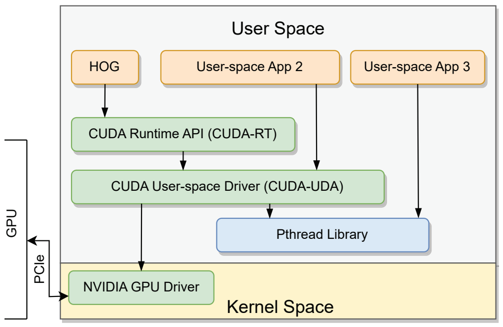


Fig. 2: CUDA architecture overview.

tracing tool that records every *transition* between kernel- and user-mode execution on every CPU core of a running computer system—all system calls and returns, interrupts/returns, faults/returns, and context switches. KUtrace also supports the tracking of user-placed markers. Post-processing turns the transitions and marker occurrences into timelines showing exactly what is executing on each core every nanosecond. KUtrace is implemented via a small number of Linux kernel patches. Total trace overhead is well under 1%, so it is usable with minimal distortion in real-time systems. Its use is described in [29] and its implementation in [28].

III. Investigation

In this section, we present an overview of the investigation we conducted to identify the source of the high latencies seen in [13] as summarized in Tbl. I. Our investigation proceeded in three phases. Initially, we sought to simply broaden the cursory investigation reported in [13], by systematically considering as many potential latency sources as seemed plausible, within HOG. However, the HOG code base proved to be so complex, it became clear to us that it would never be possible to exhaustively list (or maybe even identify) all plausible latency sources affecting it. This led us to take a new direction: finding a minimal CUDA sample program that is subject to the high latencies. This shift in focus allowed us to ultimately pinpoint SMIs as the true high-latency culprit.

A. Broadening the Initial TimeWall Investigation

In [13], the CUDA-RT functions `cudaLaunchKernel` and `cudaStreamSynchronize` were both subject to high latencies, so these functions were our initial focus. Power management and thermal throttling were already ruled out as latency sources in [13], so we did not consider them further (although the ultimate culprit proved to be "hidden" code connected to thermal issues). Although interference due to *maskable* interrupts had also been ruled out as a source, as we shall see, interrupts did become part of our investigation.

The TimeWall framework actually manages GPU kernel execution so that only one kernel may execute at a time. That is, concurrent invocations of `cudaLaunchKernel` and/or `cudaStreamSynchronize` cannot happen. However, taken together, CUDA-RT and CUDA-UDA form a complex software system that is subject to concurrency issues—like an OS, different components of this software, perhaps executing on different cores, must presumably need to update and access shared state, using locking protocols in doing so. Noting this, we initially suspected that high latencies could be caused by improper lock usage, or a bug in the lock implementation. Perhaps different components within CUDA are subject to deadlock, or a component neglects to release a lock, which is later resolved by a timeout in the locking function or some watchdog routine? This line of thinking led us to formulate the following hypothesis.

Hypothesis 1. *The high latencies observed in [13] were the result of improper lock usage in CUDA-RT or CUDA-UDA, or a bug in their underlying lock implementation.*

The high latencies seen in [13], while rare, do tend to occur regularly. Their regular occurrence led us to postulate an alternate latency source: perhaps CUDA engages in the deferred cleanup of GPU resources through garbage collection on some regular basis. This line of thinking led to the following hypothesis.

Hypothesis 2. *The high latencies observed in [13] were the result of intermittent garbage-collection operations occurring in CUDA-RT.*

Our initial investigative efforts focused entirely on attempting to validate (or refute) Hyps. 1 and 2 for the function `cudaLaunchKernel` (we had planned to consider `cudaStreamSynchronize` as well, but never actually got to it, as the latency source was later found to be not CUDA-related).

We used the same hardware configuration as in [13]. The platform we used is a Dell Precision 7920 2-socket motherboard running BIOS version 1.9.0, with an eight-core 2.10-GHz Intel Xeon Silver 4110 processor per socket, and one NVIDIA Titan V GPU. The software environment consists of the TimeWall code base (of course) running on the 5.4.0-rc7 LITMUS$^{\mathrm{RT}}$ kernel [8], [16], [17], which patches the 5.4.0-rc7 Linux kernel to allow different real-time scheduling algorithms and synchronization protocols to be defined as plugins. To ensure more predictable code execution, we disabled simultaneous multi-threading (*i.e.*, HyperThreading), directed all maskable interrupts to cores not involved in GPU work, and disabled BIOS and OS features for power-management, such as C-states and frequency scaling, as in [13]. Timer interrupts were configured to occur every 4ms (250Hz). After reproducing the TimeWall environment, we were able to run the HOG application and observe the high latencies as were observed in [13]. However, now we sometimes observed latencies of up to 21 ms! We will consider this in detail later.

Given the rare appearance of these latencies, a manual step-by-step examination of CUDA-RT instructions during execution with a traditional debugger, like `gdb`, is infeasible. Tools like `gdb` are not clairvoyant: they cannot be set to trigger *before* an event occurs. This prevented us from using breakpoints to examine program state at the time of a latency occurrence. Consequently, we leveraged the event-marker functionality of KUtrace [28], [29], also used in [13], to instrument the HOG code. Using this approach, we were able to easily identify when a high latency occurs. Whenever such a latency occurs, KUtrace provides the relevant context surrounding it, potentially enabling us to capture its cause.

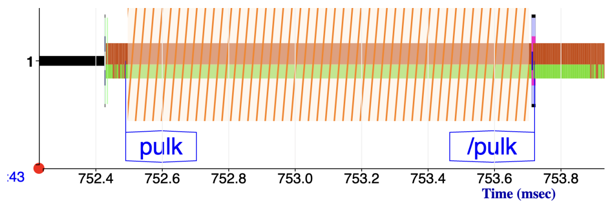
Understanding CUDA lock usage. We began our investigation by testing Hyp. 1. To that end, we needed to instrument the CUDA-RT functions with KUtrace event markers at every occurrence of a locking operation. However, each of these functions is implemented in a closed-source shared-object library (`libcudart.so`). Therefore, we could not simply inspect its source code to learn how it uses locks, or edit it to add trace markers. We needed to find a way of circumventing these closed-source limitations.

We did this by first surveying CUDA-RT's shared-object file, `libcudart.so`, for clues regarding the types of locks and the specific lock-implementation library the API uses (if any). To accomplish this, we used `ldd` to list the libraries CUDA-RT's source code dynamically links to. By means of this approach, we found the shared object `pthread.so` to be one of the API's dependencies. This led us to conjecture that CUDA-RT uses pthread locks for synchronization. An object dump of the symbols contained in `libcudart.so` confirmed that it calls the pthread mutex and read/write (rwlock) functions. As the pthread library provides both mutex and rwlocks to CUDA-RT, we proceeded to intercept the pthread locking functions for both mutexes and rwlocks to instrument them with KUtrace markers. These functions included: `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, `pthread_mutex_lock`, and and their respective unlocks.

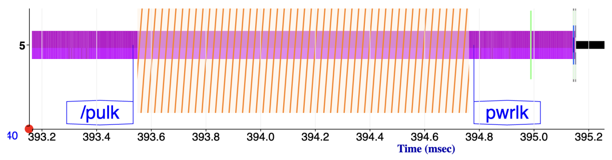
Intercepting pthread functions. To intercept calls to pthread locking functions from `cudaLaunchKernel`, we created "shim" versions of these pthread functions, and forced `cudaLaunchKernel` to call them, instead of the authentic pthread library. To implement this, we created a shared-object library containing the shim functions with identical names and argument lists as the corresponding authentic pthread locking functions. By setting the `LD_PRELOAD` environment variable to our shared-object file, we ensured that it was loaded first, thus making our shim versions of the pthread locking functions the earlier entries in the program's symbol table. This forced any call made to a genuine pthread locking function to instead call its shim counterpart.

Our shim pthread functions act as wrappers to the authentic functions. Before and after calling an original pthread function, our counterpart shim function emits KUtrace markers that output the function name and the lock address. This enabled us to precisely monitor the timing and duration of each invocation of a pthread lock inside of the CUDA-RT code. The markers also enabled us to distinguish between lock types and to identify the resources these locks protect.

Locking pattern in cudaLaunchKernel. We performed several traces of HOG using this mechanism. We saw that in the region surrounding a high-latency occurrence, `cudaLaunchKernel` runs in the only scheduled process at this



(a) High latency occurring inside of a call to `pthread_rwlock_unlock`. Shown is a timeline of CPU 1 from 752.3ms to 753.8ms within a two-minute trace. The horizontal black line is the idle process, and the multi-color horizontal bar is user-mode execution of HOG. The orange-stripe overlay is the extra ∼1ms delay. The labels `pulk` and `/pulk` are inserted into the trace by shim code, showing that the latency is inside the unlock.



(b) High latency occurring in the CUDA-RT code, between the end of an unlock call (`/pulk`) and the beginning of a write lock call (`pwrlk`), *i.e.*, outside the locking code.

Fig. 3: Two KUtrace simplified snippets showing timelines of events around an occurrence of high latency. "`pulk`" is an abbreviation for `pthread_rwlock_unlock` and "`pwrlk`" is an abbreviation for `pthread_rwlock_wrlock`.

time. This eliminated the possibility of multiple threads contending for a lock, but did not necessarily invalidate Hyp. 1, as erroneous lock usage (*e.g.*, failing to release a lock) was not precluded. We found that `cudaLaunchKernel` executes with the following lock pattern. A rwlock is write-locked using `pthread_rwlock_wrlock`, and then immediately unlocked. This rwlock lock/unlock sequence occurs 64 times, before returning.[1] In our traces, we observed two latency-occurrence categories, shown in Fig. 3. Fig. 3a exemplifies the first category: high latency occurs inside a `pthread_rwlock_unlock` call. Fig. 3b exemplifies the second category: high latency occurs inside `cudaLaunchKernel`, in-between two pthread locking functions. Based on this trace data, we made the following observation.

Observation 1. *Within `cudaLaunchKernel`, high latency can occur both inside and between pthread rwlock functions.*

From Obs. 1, high latencies can occur in-between calls to the pthread library (*i.e.*, within the CUDA-RT code itself). If Hyp. 1 were the sole latency cause, then high latency would only occur inside of pthread locking functions, not between them. Obs. 1 also has implications for Hyp. 2. If Hyp. 2 were the sole latency cause, then high latency (due to garbage collection) would only occur inside of a CUDA-RT function, not within a pthread function. This leaves us with two possibilities: either there are multiple latency sources, *i.e.*, both Hyps. 1 and 2 could be true, or high latency is caused by some activity that is asynchronous to HOG. For example, perhaps the GPU can generate a non-maskable interrupt (NMI) (*e.g.*, maybe as part of garbage collection) across some or even all CPU cores while a single CPU is executing within CUDA-RT or CUDA-UDA. We respected Occam's Razor and considered multiple error sources to be less likely, so we continued to search for a single latency source. This line of thinking led to a new hypothesis.

Hypothesis 3. *High latencies can be caused by a GPU-triggered NMI, which could be in response to some operation in `cudaLaunchKernel`.*

Narrowing down the problem scope. Since the job of CUDA-RT is to mainly perform bookkeeping, we suspected that any operation that could elicit an NMI response from the GPU would be part of the underlying CUDA-UDA. As high latencies were observed in `cudaLaunchKernel`, which wraps the CUDA-UDA function `cuLaunchKernel`, our prime suspect with respect to Hyp. 3 naturally became `cuLaunchKernel`. Accordingly, we instrumented `cudaLaunchKernel` with KUtrace markers, before and after it calls `cuLaunchKernel`. If high latency were to occur prior to the invocation of `cuLaunchKernel`, it would indicate that any operation that elicited an NMI from the GPU had occurred even before any CUDA-UDA code executes. This would refute our suspicion that code within CUDA-UDA is responsible for eliciting an NMI; it would instead point to the bookkeeping operations performed by CUDA-RT in `cudaLaunchKernel` before calling `cuLaunchKernel`.

Injecting markers. In order to surround `cudaLaunchKernel`'s calls to `cuLaunchKernel` with KUtrace markers, we first attempted to intercept calls to `cuLaunchKernel`, much like we did earlier with shim functions. However, our shim technique requires that function symbols be resolved at process initialization time. Unfortunately, `cudaLaunchKernel` dynamically searches and loads the CUDA-UDA function `cuLaunchKernel` at runtime, so this technique could not be used.

We therefore turned to a code-injection technique to insert our KUtrace markers. We added a routine at the beginning of HOG to search its address space at runtime for calls to `cuLaunchKernel`. We replaced all such calls with calls to a custom function, while still passing the original arguments meant for `cuLaunchKernel`.

The custom function was a secondary routine we created also within HOG. It simply calls `cuLaunchKernel`, forwarding the arguments it was passed, but emitting KUtrace markers before and after the `cuLaunchKernel` call. This technique allowed us to insert KUtrace markers whenever `cudaLaunchKernel` calls `cuLaunchKernel`, despite the hurdle of dynamic linking. Algs. 1 and 2 depict the effective changes in `cudaLaunchKernel`, before and after the marker injection technique was applied.

[1]These rwlock function calls actually occur while holding an outer mutex lock. We have avoided delving into that detail here to simplify the discussion.

Trace results with injected markers. We performed several traces of HOG after injecting markers. In each of these traces, we found that any high latency that occurs always happens before `cudaLaunchKernel` calls `cuLaunchKernel`. Consequently, we refined Hyp. 3 as follows.

Hypothesis 4. *High latency can occur due to the processing of a GPU-triggered NMI in response to some bookkeeping operation in `cudaLaunchKernel`, performed before it calls `cuLaunchKernel`.*

Algorithm 1 Effective logic of `cudaLaunchKernel` before marker injection

```
1: function cudaLaunchKernel
2:     PerformBookkeeping()
3:     cuLaunchKernel(SomeCUDAKernel)
```

Algorithm 2 Effective logic of `cudaLaunchKernel` after marker injection

```
1: function cudaLaunchKernel
2:     PerformBookkeeping()
3:     CustomInjectedFunc(SomeCUDAKernel)

1: function CustomInjectedFunc(args)
2:     KUtraceMarker("CUDA-UDA_start")
3:     cuLaunchKernel(args)
4:     KUtraceMarker("CUDA-UDA_end")
```

All indicators were pointing to bookkeeping instructions within CUDA-RT as the high-latency source. To confirm whether that was so, it behooved us to strip away the CUDA-RT wrapper entirely, and use CUDA-UDA directly. However, HOG is composed of almost 10,000 lines of code. Clearly, reinstrumenting it to use CUDA-UDA directly would have been a herculean task.

This motivated us to find a minimal set of CUDA instructions capable of experiencing high latency, but small enough for us to easily reimplement with CUDA-UDA directly. If the minimal program were to exhibit high latency when using CUDA-RT, but not when using CUDA-UDA directly, then Hyp. 4 would be verified. Moving to a minimal program like this proved to be a major breakthrough in our investigation. This was an important lesson learned for us in thinking about the broader issue of trying to understand the timing behavior of black-box software/hardware components.

B. Shifting to a Minimal Program

We were posed with the challenge of producing a CUDA-RT program that can exhibit high latency, using a minimal set of instructions. We had narrowed down the high-latency source to be associated with the bookkeeping code of `cudaLaunchKernel`, before it calls `cuLaunchKernel`. Thus, we started searching for a minimal instruction set by considering the characteristics of HOG we suspected could be responsible for any latency in the bookkeeping operations.

First minimal program: round-robin kernels. One property of HOG we suspected could cause high latencies in bookkeeping operations is its continual rotation through multiple kernels. We observed that every time HOG launches a kernel, it is different from the kernel it launched previously. Furthermore, we knew that CUDA code is compiled into GPU-architecture-specific microcode at runtime, using a just-in-time (JIT) compiler. To avoid incurring the overhead cost of the JIT compiler during each kernel launch, it would stand to reason that such microcode is cached. Suspecting this, we conjectured that rotating between different kernels each launch may occasionally lead to launches of kernels whose microcode is not cached. This would lead to longer-than-usual launch times, which would appear as high latencies.

Following our caching theory, we conjectured that a set of CUDA kernels, executed in a round-robin fashion, could experience high latencies. Thus, our first attempt at a minimal latency-impacted CUDA-RT program consisted of two different CUDA kernels, repeatedly executing in a round-robin fashion, with KUtrace markers placed before and after each kernel launch. This program is outlined in Alg. 3.

We immediately discovered that roughly one in 20,000 iterations of the loop in Alg. 3 incurred a 1.3ms latency. This corresponds to both the magnitude and frequency of the latencies observed in HOG as per Tbl. I. Our lucky first attempt at a minimal CUDA program had yielded a program that can experience the same high latencies as HOG! Having this program put us into a position of being able to either validate or refute Hyp. 4.

Algorithm 3 2 CUDA kernels executed back-to-back by the CUDA runtime API

```
1: function kernel1 { ... }
2: function kernel2 { ... }
3: function CUDA-RT-Min-Example
4:     for i = 1 to 100000 do
5:         KUtraceMarker("startK1")
6:         cudaLaunchKernel(kernel1)
7:         KUtraceMarker("endK1")
8:         KUtraceMarker("startK2")
9:         cudaLaunchKernel(kernel2)
10:        KUtraceMarker("endK2")
```

Algorithm 4 2 CUDA kernels executed back-to-back by CUDA-UDA

```
1: function kernel1 { ... }
2: function kernel2 { ... }
3: function CUDA-UDA-Min-Example
4:     cuCtxCreate()                    ▷ bookkeeping function
5:     LoadCUDAKernel(kernel1)
6:     LoadCUDAKernel(kernel2)
7:     for i = 1 to 100000 do
8:         KUtraceMarker("startK1")
9:         cuLaunchKernel(kernel1)
10:        KUtraceMarker("endK1")
11:        KUtraceMarker("startK2")
12:        cuLaunchKernel(kernel2)
13:        KUtraceMarker("endK2")
```

Second minimal program: CUDA-UDA-only. Given our minimal two-kernel program in Alg. 3, we had the means

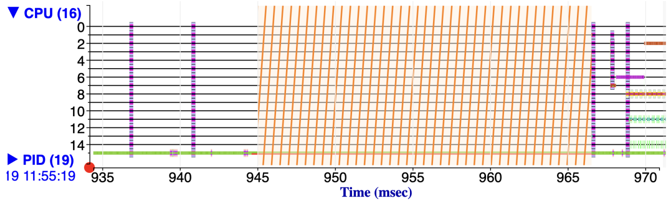


Fig. 4: A delay of 21ms (orange stripes) while running Alg. 4. Timelines for all 16 CPU cores are shown. Some OS kernel threads are executing on four cores at the far right (colors of the horizontal bars vary with process ID), and the rest are idle. The purple vertical bars are timer interrupts delivered simultaneously to multiple cores. The regular 4ms (HZ=250) spacing is lost during the delay, with five timer interrupts completely skipped and one delivered to all cores just after the unknown delay.

to verify whether CUDA-RT's bookkeeping operations are a high-latency source. To confirm whether this is so, we replaced the CUDA-RT calls with CUDA-UDA calls, and added bookkeeping functions required for CUDA-UDA-only applications. Since Alg. 3 contains only two kernel calls and just a handful of other instructions, this conversion task was trivial. Alg. 4 outlines the CUDA-UDA-only program.

Following the conversion of our minimal program to use CUDA-UDA only, we expected to validate Hyp. 4 by observing no high latencies. However, to our surprise, when we ran Alg. 4, we observed high latencies as before. They still occurred roughly one in 20,000 iterations, and lasted for about 1.3ms, *but some were as high as 21ms*, as depicted in Fig. 4. We had unexpectedly refuted Hyp. 4.

Skipping timer interrupts. Slimming down our test application from HOG to the minimal latency-impacted programs, which allowed us to capture longer trace windows, had the unintended consequence of revealing other unexpected phenomena. The first phenomenon we witnessed is that periodic timer interrupts in LITMUS$^{\mathrm{RT}}$ stopped arriving in their regular 4ms intervals during high-latency intervals. This behavior is also shown in Fig. 4.

If only one timer interrupt was due to occur within a high-latency interval, we observed that it would immediately be handled at the end of that interval, with later timer interrupts occurring as expected. However, if a high-latency interval extended over 4ms, multiple timer interrupts were skipped.

Everything stops all at once. In our traces of HOG, we only detected high latencies while HOG was the only running process in the system. However, in our traces of Alg. 4, we happened to capture instances of high latency with other processes running in the background. This led us to observe another unexpected phenomenon. When high latency occurs, tasks on other cores also become "stuck" throughout the duration of the latency. We further examined this phenomenon by retracing HOG while intentionally executing it alongside other processes. We saw from our traces, one of which is shown in Fig. 5, that when an instance of high latency occurs, non-CUDA processes across the system can also simultaneously experience high latency. This begs the question: is it reasonable to believe that a single CUDA application could effectively delay all processes in the system, including kernel threads, or is the true source of high latency something deeper in the system?

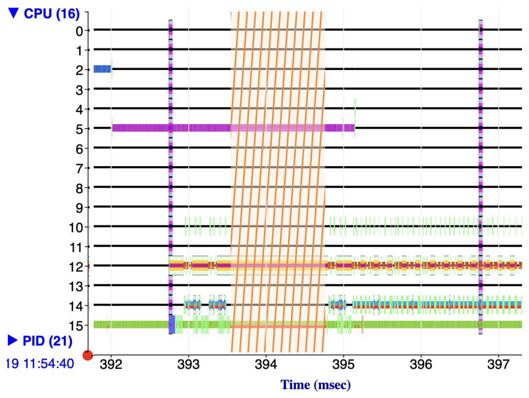


Fig. 5: Processes on all cores are affected by the high latency (orange stripes). In addition to the test-program threads running sporadically on core 5, there are unrelated processes running on cores, 12 (`in:imklog`), 14 (`rs:mainQ`), and 15 (`systemd-journal`). The 1.24ms delay here is consistent with the maximum observed HOG delays in Fig. 1.

C. Identifying the Latency Source

We were now faced with the task of finding a powerful yet silent high-latency source. The source had to be privileged enough to bring all CPUs to an abrupt halt for milliseconds at a time, yet would not appear as an event in KUtrace. As KUtrace detects events inside the OS kernel itself, this suggested a latency source even "deeper" than the OS.

System management interrupts. There are only a handful of latency sources that can evade even the OS's supervision. One such source are the system management interrupts (SMIs) and their associated system management mode (SMM).

SMIs on x86 CPUs are used to switch the CPU to critical hardware-management tasks such as thermal and power management routines. These routines are typically in the form of closed-source firmware, which handles operations relating to CPU temperature management, hardware-assisted debugging, and emulation of legacy hardware devices [22, Chapter 25]. An SMI can be invoked by external hardware (often another chip on the motherboard) via a specific input pin on the CPU, but can also be invoked by the CPU itself through writing to the CPU's model-specific register (MSR) `0xb2`.

When an SMI occurs, the CPU is preempted at the nearest instruction boundary, and is placed into system management mode (SMM) (ring -2). Inside SMM, all CPUs suspend normal operation until the hardware management routine finishes execution on CPU 0. Consequently, the execution of this routine is entirely opaque to the OS. When an SMI occurs, the OS is preempted without warning, and subsequently resumed after the SMM routine completes.

While these SMM routines should take only a few microseconds at worst, whether this expectation is met depends entirely upon the hardware's firmware developers. It is possible for the OS and all processes within to be asleep for milliseconds at a time. We therefore suspected the following.

Hypothesis 5. *High latencies are caused by excessively long SMM routines, triggered by SMIs.*

SMI Confirmation. To test Hyp. 5, we first confirmed SMIs were regularly occurring on our system. To do so, we used the `Cyclictest` package [2], which counts the number of SMI occurrences within a fixed period of time by reading a specific SMI-count MSR. `Cyclictest` revealed that our system experiences six SMIs every second. Thus, SMIs could be the high-latency culprit we had been seeking. To definitively determine whether this is so, we modified Alg. 4 to log the SMI count on each iteration of its loop. A KUtrace marker was then placed each time the SMI count increased. In the KUtrace output, we observed that high latencies were always immediately followed by an increase in the SMI count. This validated Hyp. 5, confirming that SMIs were the true high-latency culprit. However, this begged the question: what was triggering these SMIs and what SMM routine is responsible for the high latency?

Revisiting prior assumptions. Motivated by these questions, we turned our attention to investigating two common sources of SMIs: routines for triggering power management and thermal throttling. As part of this, we were forced to revisit our initial assumptions born out of the original TimeWall investigation [13]. As in that investigation, we had ensured that power-management features such as CPU frequency scaling and C-states were disabled in both the OS kernel and BIOS. However, the investigation in [13] eliminated thermal throttling as a latency culprit through the continuous monitoring of CPU frequency. This reflects the reasonable assumption that thermal throttling does not just occur "out of the blue," beyond the purview of the OS, causing CPUs to stall for up to tens of milliseconds. Thinking now that categorically discounting thermal management as a latency source might have been ill-advised, we hypothesize the following.

Hypothesis 6. *The high latency-causing SMIs are triggered by intermittent thermal-management from the motherboard.*

Temperature monitoring. To test Hyp. 6, we turned to the `lm-sensors` [9] package to monitor hardware states, such as CPU temperature and CPU fan speed. However, during the execution of Alg. 4, we saw that, in addition to the sporadic occurrence of high latency, additional high latencies coincided with `lm-sensors` querying hardware states. This discovery led us to suspect that `lm-sensors` was manually invoking SMIs. Given that the length of the high latencies during `lm-sensors`' queries are in the order of milliseconds, we hypothesized that `lm-sensors` and the mysterious SMI source were invoking related, if not the same, SMM functions.

Hypothesis 7. *The high-latency-inducing mechanism in `lm-sensors` is the same mechanism that causes the high tail latencies in HOG.*

Identifying the problem in lm-sensors. To test Hyp. 7, we first sought to identify the exact operation in `lm-sensors` that was responsible for high latencies. After consulting the `lm-sensors` documentation, we found that it reads CPU temperature and fan speed using the Linux kernel's `hwmon` sysfs interface. Specifically for our machine, this corresponds to reading from the driver-mapped files `temp2_input` and `fan2_input` from the `/sys/devices/virtual/hwmon/hwmon2` directory. To confirm that reads from these two files were indeed the problematic operations in `lm-sensors`, we executed Alg. 4 while manually reading from these files. We found that reads from `fan2_input` were able to cause high latencies in Alg. 4. We were now very close to understanding the source of the high latency-causing SMIs!

The culprit, finally! Looking for the `hwmon` device driver backing the `fan2_input` file for our machine model led us to the `dell-smm-hwmon` [6] driver, located in the Linux kernel tree at `drivers/hwmon/dell-smm-hwmon.c`. At the heart of this driver, the function `i8k_smm_func` is used to invoke SMM routines in order to query hardware states, such as fan speeds.

Due to the SMM routines in `lm-sensors` being related to CPU temperatures and fan speeds, it is highly likely that our mysterious source of SMIs is the automatic fan speed management in the motherboard firmware. From this assumption, if the default "auto" fan speed profile was causing the motherboard firmware to invoke the same SMM routines as `i8k_smm_func` to manage fan speeds, then it is likely that forcing the fans to always operate at 100% would invalidate the need for the motherboard firmware to execute these SMM routines. This would, in theory, cause high latencies in the execution of Alg. 4 to disappear. We then ran Alg. 4 with fan speed set to 100% in the BIOS, and observed no occurrences of high latencies, thus proving Hyp. 7.

Peering slightly farther into CUDA. After finding the true high-latency source, we wanted to develop a more precise understanding of where SMI/SMM-induced latencies can have an impact within the overall CUDA ecosystem. Here, the closed-source nature of CUDA eventually stymied much further progress. Still, in our experiments (and those in [13]) involving both HOG and our minimal programs, these latencies were only ever observed to impact the CUDA functions `cudaLaunchKernel` and `cudaStreamSynchronize`. Why? To shed some light on this question, we conducted further tracing experiments involving HOG and found that it spends almost all of its CPU execution time within these two functions (it invokes additional CUDA functions to transfer data to and from the GPU). The same is true of Alg. 4. As a result of these additional tracing efforts, we find it reasonable that only these two CUDA functions were ever seen to be impacted by high latencies. However, for programs that use other CUDA functions more extensively, these other functions could also similarly be negatively impacted.

Discussion. For the sake of brevity, we chose not to elaborate upon several aspects of our investigation. These include various dead ends we encountered, a few technical details pertaining to injecting KUtrace markers, and (a surprise to us) a bug we found in KUtrace. Regarding the latter, the relative rarity of the high latencies of interest resulted in a high enough volume of trace data that a buggy corner case in KUtrace was found (it has since been fixed). This experience bears witness to the importance of validating the correctness of traces, as it is easy for incorrect trace data to lead an investigation down wrong paths. To guard against this, in each step of our investigation, we corroborated our KUtrace results with results from NVIDIA's tracing tool, `nvprof`.

IV. Broader Implications of our Investigation

Our investigation revealed the sought-for high-latency source in CUDA to be SMIs inducing transitions to SMM, completely out of view of the OS kernel. This discovery raises further questions about the broader effects and consequences SMIs impose on x86-based real-time systems.

Implications of SMM latencies. While executing SMI-handler routines in SMM may be necessary to ensure the stability and reliability of the system in the long-term, their execution times can be unbounded. It is often the prerogative of the BIOS developer alone to determine and enforce an appropriate worst-case execution time (WCET) for SMI handler routines executed in SMM. Rarely are these WCETs made public. Furthermore, SMI handler routines can be invoked aperiodically. Therefore, they are notorious for creating difficulties when performing schedulability analysis of real-time task systems on x86 platforms, and are often ignored when calculating WCETs, as noted by [18].

Accounting for SMIs and SMM in schedulability analysis. As the arrival times of SMIs are unpredictable, there is no guarantee of SMI inter-arrival spacing. Thus, if we were to perform a non-probabilistic schedulability analysis that accounts for SMM execution sections, we would be required to assume the worst-case SMI arrival behavior—SMIs continuously arrive in succession, each releasing handler routines that execute up to their WCET in SMM. In such a case, the execution time of regular tasks on the OS would be delayed indefinitely, making any system unschedulable.

In order to make SMM execution sections potentially analyzable, a minimal inter-arrival time between SMIs must be enforced, and the WCETs of the SMI handler routines must be known. This would allow SMM execution sections to be modeled as m (number of CPUs) sporadic tasks with the highest priorities in the system, with periods equal to the minimal inter-arrival time of SMIs. However, such inter-arrival spacing and WCET information is not typically made public by BIOS vendors. This highlights one of the dangers of implementing real-time systems using closed-source black-box system components.

How SMIs are typically accounted for in research and industry. Despite the existence of unpredictable, non-preemptive, unbounded SMI handler routines, x86-architecture CPUs have been commonly used in real-time systems research and industry for decades. This begs the question: why are SMI-related latencies not more commonly discussed among real-time systems developers? Is everyone aware they exist?

Interested to learn how others in the real-time-systems community have handled SMI-induced latencies, we surveyed recent publications discussing SMIs or SMM, but none of them perform a schedulability analysis taking SMI/SMM delays into account [18], [19], [23], [24], [27].

Further consulting industry sources, we discovered that the detrimental effects of SMIs have been known among industrial real-time systems architects for many years. Reports published over ten years ago documented SMM latencies as long as a few milliseconds [27]. While recent SMM latencies were reported to be much lower at around $400\mu s$ in [23], they can reach as high as 177 ms under malicious interference [20]. To attenuate the effect SMM latencies have on system performance and predictability, developers in industry often use customized motherboards and BIOS firmware. This allows them to either disable SMIs, or at least know their timing parameters for schedulability analysis. For example, HP ProLiant servers, used for low-latency applications like high-frequency trading and image processing, provide a custom motherboard and BIOS that allows SMIs to be disabled [21]. We further learned that the ARM architecture also contains a processor mode similar to x86 SMM, called *Secure Monitor Mode* [14]. However, exploring this mode's impacts is outside the scope of our investigation.

Both the real-time Linux kernel patch website [4], and the Red Hat documentation website [3] mention the effects of SMIs and the difficulty of debugging them. Both also mention the near-impossibility of totally disabling SMIs, without direct contact with hardware and firmware vendors. We also located a Linux kernel bug report thread for the `dell-smm-hwmon` module, which discusses observations of a very high SMI-induced latency of approximately 500ms [5]!

Danger of closed-source black-box systems. Most x86-architecture CPUs are susceptible to SMIs. Therefore, real-time systems architects must be provided the full set of timing parameters regarding their system's hardware and software. This includes the BIOS firmware, and any hidden latency sources, no matter how paltry they may seem. If these parameters are not documented publicly, then real-time systems architects will not have the means to produce accurate system scheduling models. Left unaccounted for, lurking latency sources may manifest at the most inopportune time, leading to a catastrophic system failure.

V. Necessity of a GPU-related SMI Investigation

We have seen that the high latencies originally seen to impact HOG are actually not GPU-related at all—so why present this work in a paper that focuses on GPUs? Several GPU developer-forum posts have discussed inexplicably long

CUDA task-related latencies [7], [11], [12], of which our investigation has revealed one possible cause: SMM execution in x86 processors. To our knowledge, this latency source in the context of GPUs has not been heretofore discussed in developer forums nor in academic publications. In fact, we made our findings available to the engineers of a task-scheduler team in a major autonomous-vehicle platform supplier and they were completely unaware of any SMM-related pitfalls.

It's easy to overlook SMIs. Despite the fact that we were aware of the existence of SMIs, we (perhaps like many GPU developers) initially discounted SMIs as a latency source in our investigation, for two reasons. First, when SMIs are not considered to be under malicious interference as in [20], SMM latencies have been reported to be around 400 μs on modern hardware [23], which does not correlate with the much longer latencies we were seeing. Second, since SMIs are transparent to the OS, SMM latencies in GPU-heavy workloads can easily be mistaken as latencies either in GPU-related code itself or in the underlying highly complex framework necessary for GPU acceleration.

More broadly, these factors can greatly prolong the debugging process in the development of GPU-accelerated real-time systems, wasting both valuable time and money. We therefore wish to educate developers, especially those developing GPU applications, of the possibility of SMM-related latency issues.

VI. Conclusion

In work involving performance measurements, Ousterhout cautions to "always measure one level deeper" [26]. In this paper, we have done precisely that in trying to identify the source of the CUDA "timing glitches" seen in TimeWall and elsewhere. Specifically, we broadened the original TimeWall investigation by systematically posing several CUDA-related hypotheses. Relying mainly on KUtrace, we then carefully explained the processes we followed in verifying or refuting each hypothesis. We ultimately learned that the CUDA-related timing glitches were not a CUDA problem at all, but were due to high latencies caused by SMIs inducing transitions to SMM. The latencies due to SMIs and SMM we observed were over two orders of magnitude higher than those reported in prior publications on real-time systems. *This fact raises troubling questions regarding the viability of x86 platforms in supporting modern safety-critical real-time applications such as autonomous vehicles*.

In future work, we intend to explore the possibility of enforcing timing properties on SMIs to enable schedulability analysis that accounts for them, using open-source BIOS firmware like Coreboot [1]. Additionally, we have noted that a mode similar to SMM also exists on ARM machines [14]. We plan to conduct an experimental examination of the effects of this mode using the methods discussed in this paper.

References

[1] "coreboot: Fast, secure and flexible Open Source firmware," Online at https://www.coreboot.org/.

[2] "cyclictest - High resolution test program," Online at https://man.archlinux.org/man/cyclictest.8.en.

[3] "Debugging SMI related latencies," Online at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/system_management_interrupts.

[4] "Debugging SMI related latencies," Online at https://wiki.linuxfoundation.org/realtime/documentation/howto/debugging/smi-latency/start.

[5] "Dell inspiron 7720 fan control is freeze system by 500ms," Online at https://bugzilla.kernel.org/show_bug.cgi?id=195751.

[6] "Kernel driver dell-smm-hwmon," Online at https://docs.kernel.org/hwmon/dell-smm-hwmon.html.

[7] "LinuxCNC – Preempt-RT kernel acceptable latency," Online at https://forum.linuxcnc.org/18-computer/30418-preempt-rt-kernel-aceptable-latency.

[8] "LITMUS^{RT} home page," http://www.litmus-rt.org/.

[9] "Lm_sensors – Linux hardware monitoring," Online at https://hwmon.wiki.kernel.org/lm_sensors.

[10] "ROS: Home," https://www.ros.org/.

[11] "Too much time for kernel launch latency," Online at https://forums.developer.nvidia.com/t/too-much-time-for-kernel-launch-latency/233645/8.

[12] "Why is the kernel launch latency so high," Online at https://www.reddit.com/r/CUDA/comments/tjhdog/why_is_the_kernel_launch_latency_so_high/.

[13] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. Smith, and J. Anderson, "TimeWall: Enabling time partitioning for real-time multicore+accelerator platforms," in *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, 2021, pp. 455–468.

[14] ARM Limited, *ARM1176JZF-S Technical Reference Manual*, 2009, revision: r0p7.

[15] J. Bakita and J. Anderson, "Hardware compute partitioning on NVIDIA GPUs," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 54–66.

[16] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[17] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006, pp. 111–126.

[18] D. Dasari, B. Akesson, V. Nélis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013, pp. 39–48.

[19] B. Delgado and K. L. Karavanic, "Performance implications of system management mode," in *Proceedings of the 2013 IEEE International Symposium on Workload Characterization*, 2013, pp. 163–173.

[20] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the impact of memory errors on application performance," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 51–55, 2017.

[21] Hewlett Packard Enterprise, "Configuring and tuning HPE ProLiant Servers for low-latency applications," Tech. Rep., October 2017.

[22] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, 2008, volume 3B: System Programming Guide, Part 2.

[23] M. Madden, "Challenges using Linux as a real-time operating system," in *AIAA Scitech 2019 Forum*, 2019.

[24] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002, pp. 124–133.

[25] NVIDIA, "CUDA zone," Online at http://www.nvidia.com/object/cuda_home_new.html.

[26] J. Ousterhout, "Always measure one level deeper," *Commun. ACM*, vol. 61, no. 7, p. 74–83, jun 2018. [Online]. Available: https://doi.org/10.1145/3213770

[27] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time Linux kernel: A survey on preempt_rt," vol. 52, no. 1, feb 2019. [Online]. Available: https://doi.org/10.1145/3297714

[28] R. Sites, *Understanding software dynamics*, ser. Addison-Wesley Professional Computing Series. Boston, MA: Addison Wesley, Feb. 2022.

[29] 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*, vol. 16, no. 5, p. 54–80, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3291276.3291278