Analysis and Mitigation of Shared Resource Contention on Heterogeneous Multicore: An Industrial Case Study

Michael Bechtel
University of Kansas, USA

Heechul Yun
University of Kansas, USA

Abstract

In this paper, we address the industrial challenge put forth by ARM in ECRTS 2022. We systematically analyze the effect of shared resource contention to an augmented reality head-up display (AR-HUD) case-study application of the industrial challenge on a heterogeneous multicore platform, NVIDIA Jetson Nano. We configure the AR-HUD application such that it can process incoming image frames in real-time at 20Hz on the platform. We use micro-architectural denial-of-service (DoS) attacks as aggressor tasks of the challenge and show that they can dramatically impact the latency and accuracy of the AR-HUD application, which results in significant deviations of the estimated trajectories from the ground truth, despite our best effort to mitigate their influence by using cache partitioning and real-time scheduling of the AR-HUD application. We show that dynamic LLC (or DRAM depending on the aggressor) bandwidth throttling of the aggressor tasks is an effective mean to ensure real-time performance of the AR-HUD application without resorting to over-provisioning the system.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Industrial Challenge, Real Time, SLAM, Microarchitectural DoS Attack

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Heterogeneous multicore computing platforms are increasingly utilized in safety-critical cyber physical systems (CPS) as they can offer significant performance improvements while simultaneously meeting size, weight, and power (SWaP) constraints. However, contention for shared microarchitectural resources, such as shared cache, between the computing elements in such a platform remains a significant challenge for system predictability. In particular, shared resource contention can impact the execution timings of critical real-time tasks and thus jeopardize the safety of the CPS. Moreover, such contention can also be intentionally induced by malicious third-party actors with the goal of compromising the performance and safety of CPS. These types of adversaries have come to be known as microarchitectural Denial-of-Service (DoS) attacks [7], and are especially problematic for high-performance CPS that need to run multiple concurrent applications. Given the recent trends towards cloud-based continuous software delivery in CPS, as can be seen with ARM's SOAFEE initiative [3] for the automotive industry, it is conceivable that such DoS attacks could be remotely deployed on future CPS.

Understanding and addressing shared resource contention in multicore has been of intense interest for both academia and industry in recent years. In particular, ARM issued an Industrial Challenge in ECRTS 2022 with the goal of stimulating new approaches and techniques to address the problem of shared resource contention [2]. The challenge is centered around an augmented reality head-up display (AR HUD) case-study for automotive applications. The case-study application is composed of two main components: a Visual Simultaneous Localization and Mapping (SLAM) task [13], and a DNN-based driver head pose estimation task [14]. The SLAM task is composed of three main threads, all of which run on the CPU, whereas the DNN-based head-pose estimation task (we henceforth refer it as the DNN task) may utilize the GPU. The application represents a mixed-criticality system that must leverage high-performance heterogeneous multicore embedded platforms. As such, the challenge seeks to find ways to analyze and optimize performance bounds of such critical real-time tasks even in the presence of "aggressor tasks", which may contend with the critical real-time task in accessing shared resources (i.e., microarchitectural DoS attacks).

In this paper, we study the impact of shared resource contention to the performance of the AR HUD case-study application of ARM's Industrial Challenge. Through our study, we aim to answer the following questions: (1) Can we safely consolidate the two real-time tasks with mixed criticality (the SLAM algorithm and head pose detection) in the AR-HUD case-study on a representative heterogeneous system-on-chip (SoC) processor and achieve good real-time performance? (2) Does shared resource contention between the two AR HUD tasks impact their respective performance? E.g., the accuracy of the obtained position/trajectory estimate of the SLAM task. (3) Can we guarantee real-time performance of the AR-HUD application in the presence of aggressor tasks, which may be maliciously designed to cause high shared resource contention, without excessive over-provisioning?

To answer these questions, we systematically conduct experiments on an NVIDIA Jetson Nano, a representative heterogeneous embedded multicore platform that features a quad-core ARM Cortex-A57 CPU and an integrated NVIDIA GPU. Our findings are as follows: We were able to configure the AR-HUD application such that it can process incoming image frames in real-time at 20Hz. However, we find that shared resource contention can significantly impact the accuracy of the SLAM algorithm, which results in significant deviations of the estimated trajectories from the ground truth even when it could process all input image frames in real-time. We find that a certain type of DoS attack [6], when added to the system as aggressor workloads, is especially effective in increasing inaccuracies in the SLAM generated trajectory, and in increasing the execution times to a degree that it completely fails to meet the real-time requirements. For example, when the DoS attack tasks were scheduled as best-effort (non-RT) tasks together to fully load the system, the SLAM task was unable to keep up with the input sensor data, and can only successfully process $\sim$26% of the input frames, resulting in a complete system failure.

We then explore a mitigation method to help protect the AR-HUD real-time application from the micro-architectural DoS attacks. In particular, we evaluate a bandwidth throttling approach that selectively throttles best-effort tasks in order to protect the real-time tasks from shared resource contention. To accomplish this, we use the RT-Gang scheduling framework [1], which implements DRAM bandwidth throttling of best-effort tasks whenever any real-time task is running. At the same time, if no real-time task scheduled on the system, RT-Gang gives best-effort tasks full bandwidth, allowing for improved system utilization during slacks between real-time tasks. We made two modifications to the vanilla RT-Gang to enable Last-Level Cache (LLC) bandwidth throttling and to support multiple concurrent real-time (gang) tasks. These changes allowed us to safely consolidate the AR-HUD application even in the presence of malicious DoS attacks.

This paper makes the following **contributions**:

- We systematically study the effect of shared resource contention on the performance of the AR HUD application, using state-of-the-art denial-of-service (DoS) attackers as additional aggressor tasks [7]. We find the recently proposed LLC bank-aware DoS attack [6] to be especially effective in delaying the execution latency and, consequently, degrading the accuracy of the SLAM algorithm. We also find that executing GPU kernels can significantly impact the performance of the SLAM algorithm, which runs on the CPU.

- We show that the SLAM algorithm's accuracy can degrade even when input frames are processed in real-time (i.e., meeting their deadline) because the increased execution latency of the SLAM task's key threads can significantly impact the accuracy. We find that these delays cause the trajectory generated by the SLAM algorithm to significantly deviate from the ground truth.
- We evaluate a LLC bandwidth throttling approach that dynamically throttles best-effort (non RT) tasks to protect real-time tasks (the AR-HUD application). We show that our mitigation approach can allow the SLAM algorithm to operate in real-time even in the presence of DoS attacks.

The rest of this paper is organized as follows. Section 2 describes the ARM Industrial Challenge problem. Section 3 describes microarchitectural DoS attacks for the challenge. Section 4 discusses our experimental setup. Section 5 presents our empirical evaluation of the challenge's case-study application. Section 6 presents a mitigation approach and its effects. We review related work in Section 7 and conclude in Section 8.

2 ARM Industrial Challenge 2022: Augmented Reality Head-Up Display (AR-HUD) Application

Addressing the impacts of shared resource contention is of critical importance in many high-performance CPS, such as those in the robotics and automotive fields. This is especially the case given the increased importance of consolidating high-performance mixed criticality applications in CPS. To stimulate further research on this topic, ARM introduced an Industrial Challenge in ECRTS 2022. The challenge presented an augmented reality head-up display (AR HUD) application in the automotive context as a case-study. As an advanced driver assistance system (ADAS), this application provides additional alerts and notifications to the driver of a vehicle in real-time. In particular, these alerts are overlaid on real-world objects using augmented reality (AR) technology. For the suggested AR HUD application, it is mainly comprised of two components: a Visual SLAM task, and a head pose estimation task. We now briefly introduce and discuss both AR HUD components.

For many autonomous cyber physcial systems (CPS), localization and 3D map generation are important steps for real-world performance. Increasingly, many CPS employ Simultaneous Localization and Mapping (SLAM) algorithms to perform both operations in a single step. In a SLAM algorithm, input sensor data is received and utilized to both estimate a system's current position in, and generate/update a 3D map of a given environment. Vision (camera) and range sensors such as LIDARs, lasers and sonars can be used for SLAM. This category of SLAM algorithms that utilize vision has come to be known as *Visual SLAM*.

In the ARM industrial challenge, the OV^{2}SLAM algorithm [13] is suggested as part of the AR HUD case study [2]. OV^{2}SLAM is a Visual SLAM algorithm that is geared towards real-time applications and emphasizes processing time in addition to SLAM performance. It is composed of four main components with each one being assigned to a separate thread:

1. The *Front-End* thread performs real-time pose estimation of the camera sensor. It is also responsible for creating the keyframes used to generate 3D maps of surrounding environments. Note that this thread runs for every input frame that is received, meaning that it is a periodic task in nature. For our purposes, we target a per-frame deadline of 50 ms as the input datasets we use in our evaluations playback data at a frequency of 20 Hz.

2. The *Mapping* thread uses keyframes generated in the *Front-End* to generate new 3D map points. It primarily does this by performing triangulation on the keyframes. Then, if a new keyframe has not arrived, it will also perform local map tracking in order to minimize drift. Unlike the *Front-End*, the *Mapping* thread is aperiodic as it is event-driven and only runs when a new keyframe is generated.
3. The *State Optimization* thread performs two main operations. First, it runs a local bundle adjustment (BA) to refine camera pose estimations. Second, it runs a keyframe filtering pass that prevents redundant keyframes from being processed in future BA operations. Note that this thread is also aperiodic as it relies on input from the *Mapping* thread, meaning that it is also event-driven by proxy.
4. The *Loop Closer* thread performs an online bag-of-words (BoW) operation to detect loop closures in a system's given trajectory. However, we do not employ this thread in our case study as it is not necessary for the target AR HUD application [2].

Note that only the *Front-End* thread runs for every input frame that is fed to OV^{2}SLAM. The remaining threads will then only run when necessary, such as when a new keyframe is created.

By default, the OV^{2}SLAM algorithm can be run in one of three different modes: *accurate*, *fast*, and *average*. The *accurate* mode of operation performs all four steps described above, including Loop Closure, and is intended to maximize accuracy while still maintaining a control frequency of 20 Hz. On the other hand, the *fast* mode of operation instead sacrifices some accuracy so that it can operate at a much faster 200 Hz control frequency. To achieve this, the *fast* version uses a faster (but less accurate) keypoint detection algorithm and does not perform the Loop Closure step. The *average* version then operates in between the other two versions performance-wise. In other words, it runs at a control frequency between 20 and 200 Hz, and achieves accuracy worse than the *accurate* version but better than the *fast* version. Like the *accurate* version, though, the *average* version also performs Loop Closure. The Industrial Challenge suggests to use the *fast* version for its superior real-time performance and good accuracy.

For the AR HUD application, it is also important that the ADAS alerts provided to the driver are displayed in a way that matches the driver's viewpoint. To achieve this, the head pose of the driver can be estimated so that the AR display can be corrected as necessary. As such, the AR HUD application employs a head pose estimation task for its second component. The Industrial Challenge suggests to use the HopeNet-Lite head pose estimator [14], as it can run in real-time on many embedded heterogeneous multicore platforms. We further discuss this component in our evaluation setup.

3 Microarchitectural Denial-of-Service (DoS) Attacks

As part of the Industrial Challenge, ARM also envisioned the presence of *aggressor workloads* in the AR-HUD case study [2]. These aggressor workloads may be co-scheduled alongside the AR HUD application and may cause shared resource contention.

For our aggressor workloads, we employ microarchitectural denial-of-service (DoS) attacks that have been described in literature [36, 7, 6]. These DoS attacks target various microarchitectural resources in multicore platforms (e.g. LLC, DRAM) and can cause significant execution time delays to cross-core real-time tasks, even if they run on dedicated cores and have dedicated LLC partitions. In this work, we want to know the impacts such DoS attacks can have on the AR HUD application. The specific DoS attacks we employ in our evaluations are as follows:

- The *bandwidth* benchmark from the IsolBench suite [36]. This benchmark is designed to perform continuous accesses to a target shared resource (e.g. LLC or DRAM) in a sequential manner. To be more specific, it performs sequential accesses over a 1D array at a cache line granularity (i.e. all accesses are 64B apart). We refer to DoS attacks based on this benchmark as *Bw*.
- The *latency-mlp* benchmark from the IsolBench suite [36]. Much like the *bandwidth* benchmark, *latency-mlp* continually accesses a target resource but differs in its access pattern due to its pointer chasing nature. Namely, it performs random accesses over multiple parallel linked lists (PLL). We refer to DoS attacks based on this benchmark as *PLL*.
- The cache bank-aware attacks from [6]. Much like memory-aware attacks [5], these attacks based on the *PLL* attacks above, but are modified to only access a specific cache bank in order to generate maximum cache bank contention in accessing the LLC. We refer to this attack as *BkPLL*.

Furthermore, these DoS attacks can be configured in two additional facets. First, they can all be configured to perform either read or write accesses. As such, we test DoS attacks of both access variations in our testing. Second, as mentioned above, the attackers can be configured to access the LLC or DRAM, so we employ separate DoS attacks targeting each shared resource. Note that we configure the *BkPLL* attacks to only access the LLC, as they are specifically designed for that resource. Putting it altogether notation wise, we use the following naming convention for DoS attacks:

<DoS attack type><access type>(<target resource>)

Where *DoS attack type* is one of the attacks from the above list, the *access type* is either read or write, and the *target resource* is either the LLC or DRAM. For example, an instance of the *Bw* attack that performs read accesses targeted to the LLC would be referred to as *BwRead(LLC)*. In total, we employ 10 different DoS attacker tasks in our evaluation.

4 Experiment Setup

In this section, we describe the experimental setup for our case study of the AR HUD application.

4.1 Hardware Platform

We deploy all target applications and DoS attacker tasks on an ARM-based Nvidia Jetson Nano embedded platform. The Jetson Nano equips a quad-core cluster of Cortex-A57 cores, with each core having its own private L1 instruction and data caches and all cores sharing access to a global L2 cache. Table 1 shows the basic characteristics of the Jetson Nano platform.

4.2 Application Setup

As discussed in Section 2, the AR HUD application is comprised of two main components: the OV^{2}SLAM Visual SLAM task, and the HopeNet-Lite head pose estimation task.

For the SLAM task, we use the *fast* setting of OV^{2}SLAM, as recommended in the Industrial Challenge [2], which is comprised of three threads: *Front End*, *Mapping*, and *State Optimization*. The *Front End* thread is invoked whenever the camera provides a new image frame, which in our case is at a fixed rate of 20Hz. The *Mapping* and *State Optimization*

Platform	Nvidia Jetson Nano
SoC	Tegra X1
CPU	4x Cortex-A57 @ 1.43GHz
GPU	128-core Maxwell
Shared LLC (L2)	2MB (16-way)
Memory (Peak B/W)	4GB LPDDR4 (25.6 GB/s)

Table 1 Nvidia Jetson Nano hardware specifications.

threads are invoked conditionally when the *Front End* thread decides to generate a new key frame (See 2 for more details). As the SLAM task is *real-time critical* [2], we use the Linux `SCHED_FIFO` real-time scheduler and assign it a real-time priority of 2. Furthermore, we assign all three threads of the SLAM task onto two CPU cores, Core 0 and 1, which we experimentally determined to be sufficient for accurate and timely SLAM processing—when they run in isolation. Note that the maximum observed per-core CPU utilization of the SLAM threads is less than 70%, meaning there is additional slack that can potentially be used by best-effort tasks. In our study, we use the DoS attacks (Section 3) as best-effort tasks. Lastly, while the original OV^{2}SLAM release is based ROS1, we ported it to ROS2 as the latter better supports real-time scheduling and does not require an additional process (e.g., roscore) for communication.

For our testing of the SLAM task, we use the five Machine Hall (MH) scenarios from the EuRoC dataset [9], MH01-MH05. We provide image frames to the SLAM task through an instance of `rosbag2` [15] running on Core 2, also with a real-time priority of 2 such that it is not delayed by any best-effort tasks. We find that the overhead of `rosbag2` is about ∼5-10% in core utilization. Note that all MH datasets playback data at a frequency of 20Hz, which we use for the target SLAM frequency.

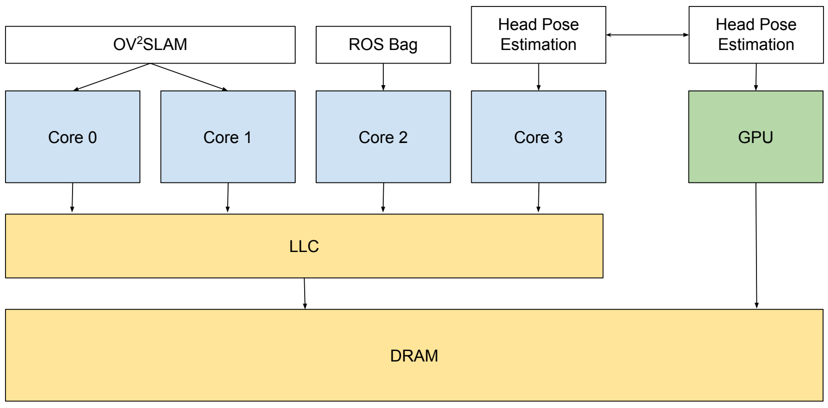


Figure 1 Tasks to core assignments of the AR-HUD case-study on Jetson Nano.

For the second component of the AR HUD application, the head pose estimation task, we use the HopeNet-Lite DNN model [14], which is a lightweight version of the original HopeNet model [32] and uses the ShuffleNet V2 [24] as its backbone. Note that the HopeNet-Lite is mainly processed by the GPU but a CPU core is used to launch the GPU kernel and monitor its progress. In our testing, the HopeNet-Lite model could process input frames at ∼35 ms per frame when running in isolation on the Jetson Nano platform. We configure the task to run periodically at the same 20Hz rate as the SLAM task and assign it a real-time priority of 1 as the task is determined to be a *non-critical, high priority task* [2]. In addition, we pin the HopeNet-Lite task to a CPU core distinct from the SLAM task cores, Core 3, so that none of its required CPU operations (e.g. CUDA kernel launch, etc.) are interfered with by the OV^{2}SLAM threads.

Figure 1 gives a visual representation of the setup we use and how we assign the AR HUD tasks to CPU cores on the target platform.

Task	Thread	Cores(s)	Real-Time Priority	Rate (Hz)
OV^{2}SLAM	Front-End	0,1	2	20
	Mapping			-
	State Optimization			-
EuRoC playback	ROS bag	2	2	20
Head Pose Estimation	HopeNet-Lite model	3,GPU	1	20

Table 2 Real-time tasks/threads/core mapping and scheduling parameters in the AR HUD case study. Note that all real-time tasks are scheduled using the `SCHED_FIFO` real-time scheduler.

Table 2 then shows the real-time characteristics for all of the tasks and threads we utilize in the AR HUD case study.

4.3 Operating System Setup

For the operating system we run Ubuntu 18.04 with Linux kernel 4.9, which is patched with PALLOC [42] to support LLC partitioning. PALLOC exploits virtual memory page translations to enforce page allocations to specific page colors. With PALLOC, we partition the LLC into four equally sized partitions (colors) and perform a 2/2 split of those partitions. Namely, the OV^{2}SLAM algorithm gets two LLC partitions, and all other tasks share the remaining two cache partitions. Note that all best-effort tasks—those that are scheduled using Linux's default CFS scheduler—also share the latter two cache partitions in order to minimize any performance impact to the SLAM task, which is real-time critical. Note that, in PALLOC, tasks—not cores—can be mapped to any cache partitions.

5 Analyzing the Effects of Shared Resource Contention

In this section, we evaluate the impact of shared resource contention on the performance of the OV^{2}SLAM algorithm.

5.1 Impact of Co-scheduling DoS Attacks

In this experiment we evaluate the impacts of DoS attack co-runners and whether they are effective in degrading OV^{2}SLAM performance in a given scenario. Note that we did not execute the HopeNet-Lite DNN task in this experiment in order to focus on SLAM performance and its sensitivity to DoS attacks.

The experiment setup is as follows: We first run an instance of the *fast* OV^{2}SLAM algorithm on the MH01 scenario from the EuRoC dataset [9]. Once finished, we calculate the algorithm's *Absolute Trajectory Error (ATE)* relative to the known ground-truth trajectory [9]. We then repeat the experiment but with instances of a DoS attacker on all four available cores. We again calculate the ATE and compare it to the solo case to determine whether the attackers had any noticeable impact.

Note that we run the DoS attacks as best-effort tasks. Because Linux strictly prioritizes real-time tasks over best-effort ones, the DoS attack tasks can only be executed on cores when they are not executing the RT tasks. In other words, whenever the RT threads (the SLAM task) become ready, they immediately preempt any DoS attacker tasks. Note also that, as mentioned earlier, the DoS attack tasks are assigned to a separate LLC cache partition from the SLAM task, which further minimizes any negative effect of co-scheduling.

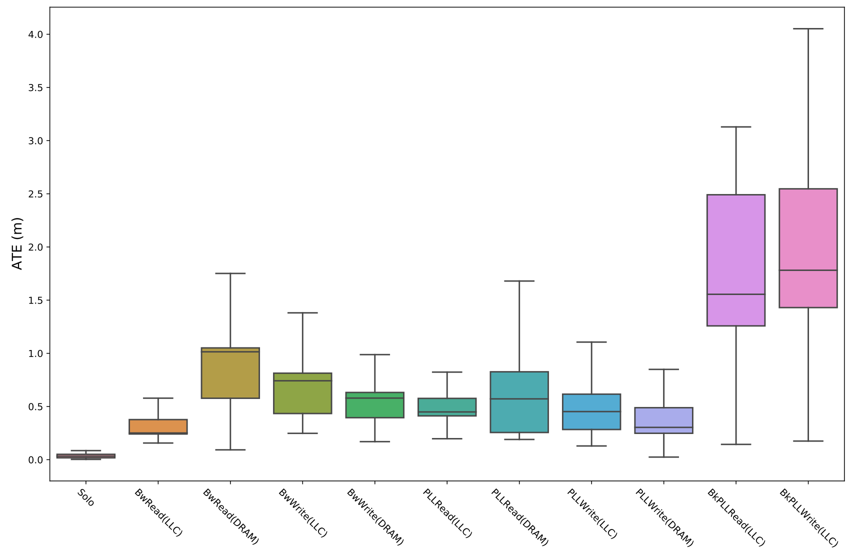
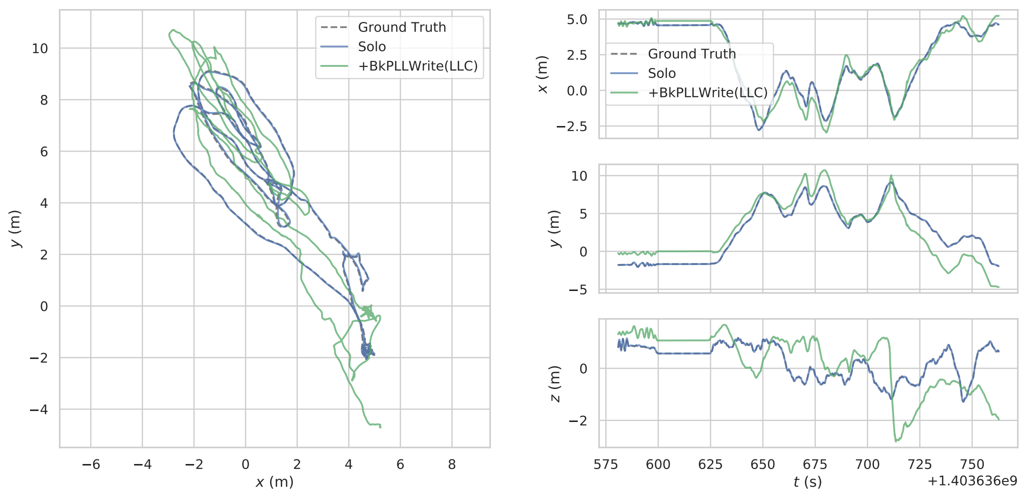


Figure 2 Impact of DoS attacks on the ATE of the OV^{2}SLAM algorithm.

Figure 2 shows a boxplot of the OV^{2}SLAM ATE, collected over the entire duration of the MH01 dataset, both alone and alongside each of the tested DoS attacks (X-axis). First and foremost, we find that all of the tested DoS attacks cause significant increases to the tracking performance, despite the fact that they cannot preempt the real-time SLAM task running on the same cores, due to contention from remote cores. Note that ATEs of ∼0.3 or more means significant deviations from the ground truth, which could potentially cause system failure (e.g., a crash) in real-world CPS [22]. Moreover, we find that the recently proposed cache bank-aware attacks (denoted as BkPLLRead(LLC) and BkPLLWrite(LLC) for read and write, respectively) [6] are by far the most effective in terms of ATE increase. Concretely, *BkPLLRead(LLC)* attack increased the median ATE to >1.7 (∼49X increase over solo) and the *BkPLLWrite(LLC)* attack increased it to >1.9 (∼55X increase). In other words, these attacks caused the OV^{2}SLAM algorithm's detected trajectory to deviate from the ground truth trajectory by close to two meters on average, and more than four meters in the worst-case.

Figure 3 shows the trajectory generated by OV^{2}SLAM, and its X, Y and Z positions over time, when run alongside the *BkPLLWrite(LLC)* attackers, and how it compares to the ground truth trajectory. As shown clearly in the figure, even though the attackers do not alter CPU scheduling of the SLAM task, they still manage to incur massive impact in the algorithm's accuracy due to shared resource contention. In this particular attack, the source of contention is bank-level contention in the LLC, which cannot be prevented by standard cache partitioning techniques, like page coloring [6]. However, other DoS attacks that target



(a) Trajectory in XY plane **(b)** X, Y and Z positions over time

Figure 3 OV^{2}SLAM trajectories generated alone and against *BkPLLWrite(LLC)* attackers.

different shared resources (e.g., DRAM) are also effective, albeit to a less degree, as indicated in Figure 2.

To understand why these DoS attacks are effective, we measure the per-second LLC and DRAM bandwidths consumed by the OV^{2}SLAM task over the course of its runtime when it runs alone in isolation. Note that, according to [6], cache bank-aware DoS attacks are most effective against LLC-sensitive workloads.

Figure 4 shows the measured LLC and DRAM bandwidths. Surprisingly, the OV^{2}SLAM is not an overly resource hungry task as, on average, it consumes less than a gigabyte per second of either LLC or DRAM bandwidth. This suggests that even modestly resource demanding applications can still significantly suffer from shared resource contention that cannot be mitigated by the existing Linux real-time scheduling framework.

5.2 Impact of Co-scheduling HopeNet-Lite on the Integrated GPU

In this experiment, we evaluate the impact of co-scheduling the HopeNet-Lite head pose estimator on the performance of the OV^{2}SLAM task. We compare the following configurations: *+DNN* in which HopeNet-Lite is co-scheduled with OV^{2}SLAM; *+DoS* in which four instances of the *BkPLLWrite(LLC)* DoS attack are co-scheduled with OV^{2}SLAM; and *+DNN&DoS* in which both HopeNet-Lite and the DoS attack tasks are co-scheduled with OV^{2}SLAM. Note that the DoS attackers are best-effort tasks while both OV^{2}SLAM and HopeNet-Lite are real-time tasks. Based on the results from Figure 2, we only test the *BkPLLWrite(LLC)* DoS attack in these experiments as it is the most effective at degrading the performance of the SLAM algorithm.

Figure 5 shows the generated trajectories for both the *+DNN* and *+DNN&DoS* cases. First, the addition of the HopeNet-Lite task significantly impacts the accuracy of OV^{2}SLAM with the median ATE increasing to >0.8, which can be seen by how the trajectory deviates from the ground truth. However, when HopeNet-Lite is combined with the DoS attacks, in *+DNN&DoS*, OV^{2}SLAM suffers a drastic performance degradation to the point that it completely fails to generate a full trajectory. This is because OV^{2}SLAM is unable to keep up

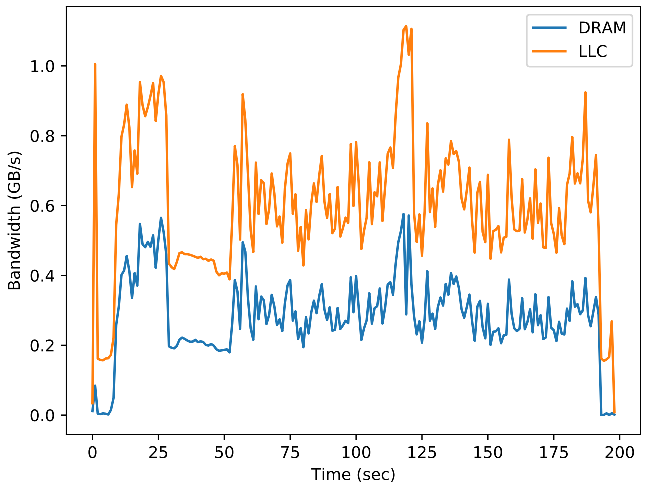
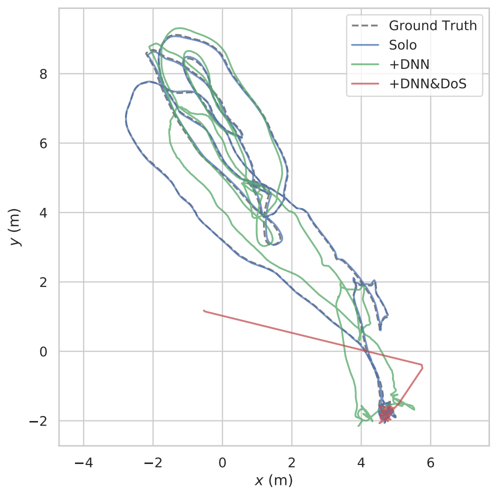
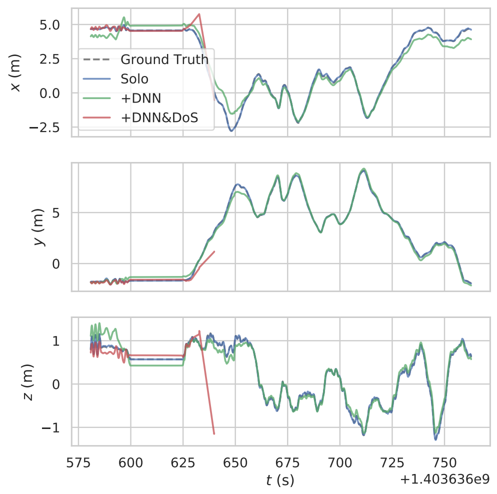


Figure 4 LLC and DRAM bandwidth consumed by OV^{2}SLAM over the course of its runtime. Note that OV^{2}SLAM is run alone without any resource intensive co-runners.



(a) Trajectory in XY plane



(b) X, Y and Z positions over time

Figure 5 OV^{2}SLAM trajectory when run alongside both *BkPLLWrite(LLC)* DoS attackers and a GPU-based DNN application.

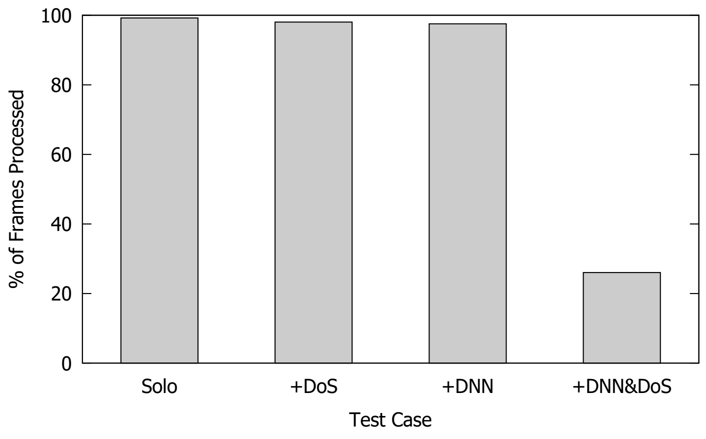


Figure 6 Percentage of input frames processed by OV^{2}SLAM with resource intensive CPU and GPU co-runners.

with the input data—due to shared resource contention—and ends up dropping a majority of the frames. Our analysis shows that only ∼26% of the input image frames were processed in *+DNN&DoS*, as opposed to more than >97% in either the *+DNN* or *+DoS* cases. Figure 6 shows the percentage of frames processed by the OV^{2}SLAM instance.

To determine whether this GPU interference can be alleviated, we next test whether reducing the scheduling frequency of the HopeNet-Lite task mitigates the contention it causes. To that end, we re-run the *+DNN* test scenario but vary the frequency of the HopeNet-Lite task from 25 Hz to 5Hz, in decrements of 5 Hz, and measure OV^{2}SLAM's ATE in each case.

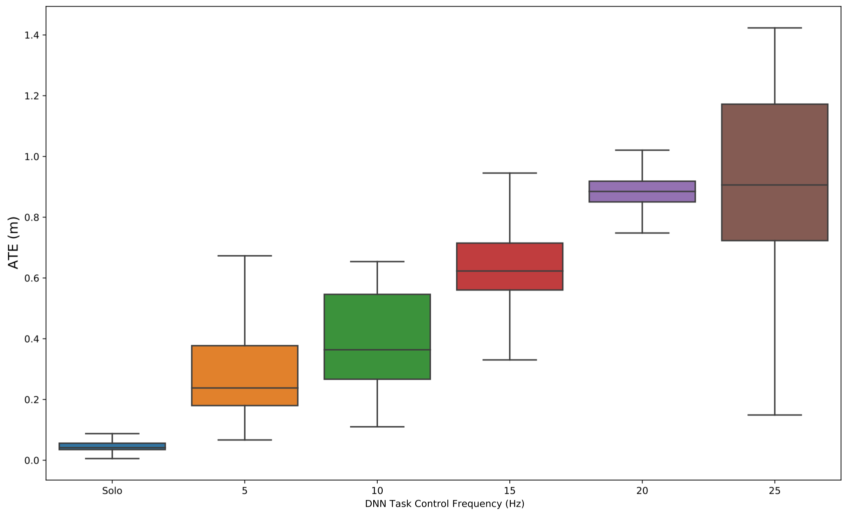
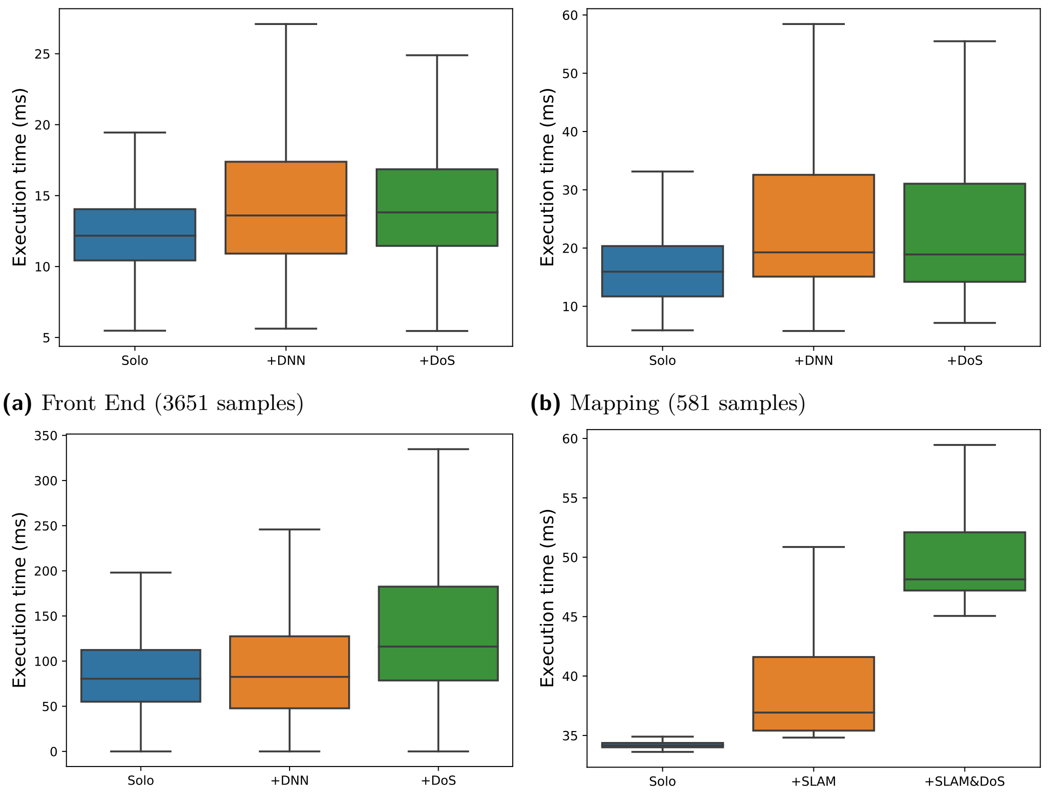


Figure 7 Impact of head pose estimation control frequency on OV^{2}SLAM ATE performance.

Figure 7 shows the results for the different scheduling rates. First, we find that lower frequencies have less impact on ATE performance, largely because the GPU causes interference less often. That being said, we do see non-trivial performance loss in all scenarios. Even in the 5 Hz scenario, the median ATE is still increased by ∼8X, from ∼0.03 in *Solo* to ∼0.24 in *+DNN*.

5.3 Runtime Analysis of OV^{2}SLAM and HopeNet-Lite

To further investigate the impacts of shared resource contention on the AR HUD application, we perform a detailed execution time analysis on the three CPU threads of the OV^{2}SLAM task—the *Front End*, *Mapping* and *State Optimization* threads—and the DNN-based head pose estimator HopeNet-Lite. For the OV^{2}SLAM threads, we measure and record the execution times of each thread when they perform their main computational loop (e.g. *Front End* receives a new input frames, *Mapping* receives a new keyframe, etc.). For the DNN task, we measure its inference times across 1000 input frames. We then compute the distribution of execution times for all tasks and threads to determine whether any of them experience execution delays due to co-runner interference. For the OV^{2}SLAM analysis, we re-run three of the test scenarios previously performed: the *Solo* case where it runs alone, the *+DoS* case alongside *BkPLLWrite(LLC)* DoS attack, and the *+DNN* case alongside the HopeNet-Lite task. For the HopeNet-Lite task, we also measure its inference times in three different scenarios: *Solo* when it runs alone, *+SLAM* when run with the SLAM task, and *+SLAM&DoS* when run with both the SLAM task and *BkPLLWrite(LLC)* DoS attack.



(a) Front End (3651 samples)   **(b)** Mapping (581 samples)

(c) State Optimization (581 samples)   **(d)** Head Pose Estimation (1000 samples)

Figure 8 Execution time distributions of all real-time AR HUD tasks and threads.

Figure 8 shows the execution time distributions for all real-time tasks and threads in each of their respective scenarios. For the OV2SLAM task, we observe execution time increases in all three threads when adversarial co-runners were present. Notably, though, we find that the *Front End* and *Mapping* threads see greater WCET slowdowns from the DNN co-runner than they do the DoS attacks. On the other hand, the *State Optimization* thread is much more impacted by the DoS attacks in terms of the observed WCET. Given the disparity in SLAM ATE loss between the DoS attackers (55X) and the DNN co-runner (27X), we find that *State Optimization* is of greater importance for OV^{2}SLAM in generating more accurate trajectories, which is consistent with prior findings [22].

Lastly, Figure 8d shows the time distributions for the HopeNet-Lite task. Similar to the SLAM task, the HopeNet-Lite task is also slowed down when the two are run together (on different cores/GPU), going from an average inference time of ∼34 ms alone to ∼37 ms alongside the SLAM task. When the DoS attacks are then added, the average inference time increases again to ∼48 ms.

6 Mitigating Shared Resource Contention Using RT-Gang

In this section, we explore potential migration methods to protect the real-time AR-HUD application in the presence of aggressor tasks (i.e., DoS attacks).

6.1 LLC Bandwidth Throttling of Best-effort Tasks

As previously seen, when DoS attackers (e.g., BkPLLWrite) are present, the performance of the OV^{2}SLAM algorithm is significantly reduced even though the DoS attackers cannot preempt the SLAM task, as the former are best-effort tasks and the latter is a real-time task. This is because existing real-time schedulers in Linux are unable to protect against shared resource contention from best-effort tasks on different cores.

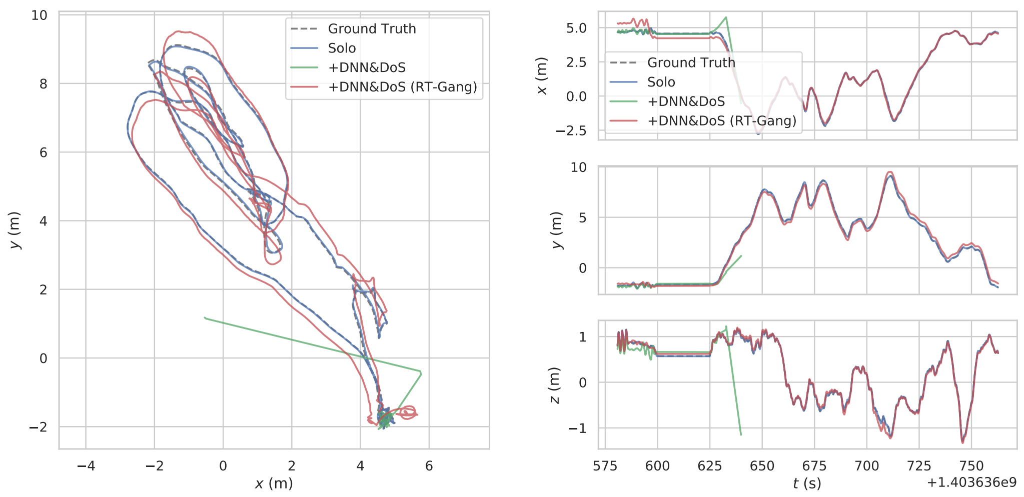
In this work, we use the RT-Gang scheduling framework [1], which is designed to protect safety-critical real-time tasks against resource intensive best-effort tasks. RT-Gang supports a simple real-time gang scheduling policy, which allows only one parallel real-time gang at a time across all cores. Moreover, RT-Gang supports memory bandwidth throttling of best-effort tasks to protect any currently running real-time gang task. In other words, RT-Gang throttles any cores that execute best-effort tasks whenever a real-time gang task is running on any cores in the system. On the other hand, if no real-time gang is scheduled on the system, then the best-effort tasks have full access to the memory bandwidth. In our case study, we use DoS attack tasks as best-effort tasks that cause contention on the shared resources. As such, we can use RT-Gang's ability to dynamically throttle best-effort tasks to help protect the AR-HUD application.

We make two *extensions* to the vanilla RT-Gang in order to better support the needs of the AR-HUD application case-study: (1) we add support for multiple concurrent real-time gang scheduling capability. This was needed to assign the OV^{2}SLAM and the HopeNet-Lite on two separate real-time gangs while allowing them to be scheduled together simultaneously on different cores. (2) We modified the memory bandwidth throttling capability to support LLC bandwidth throttling. This was needed as our best DoS attack, BkPLLWrite, targets the LLC and does not consume any DRAM bandwidth [6].

6.2 Impact of Using Extended RT-Gang

Using the extended RT-Gang, we repeat the experiment in *+DNN&DoS* but with RT-Gang enabled, which we label as *+DNN&DoS (RT-Gang)*, to see whether RT-Gang can effectively protect the performance (accuracy) of the OV^{2}SLAM algorithm.

Figure 9 shows the trajectory generated by the OV^{2}SLAM task, and the X, Y and Z positions of the trajectory over time. Compared to *+DNN&DoS* case, where OV^{2}SLAM



(a) OV^{2}SLAM trajectory in the *+DNN&DoS* case w/ and w/o RT-Gang enabled. **(b)** X, Y, and Z positions of the OV^{2}SLAM trajectories over time.

Figure 9 Impact of RT-Gang on OV^{2}SLAM performance.

could only create a highly inaccurate partial trajectory, RT-Gang effectively protects against the *BkPLLWrite(LLC)* attackers such that OV^{2}SLAM can produce a trajectory that is significantly closer to the ground truth. Likewise, Figure 10 shows the percentage of frames processed with our extended version of RT-Gang enabled. Note that the OV^{2}SLAM is able to process >96% of input frames with RT-Gang.

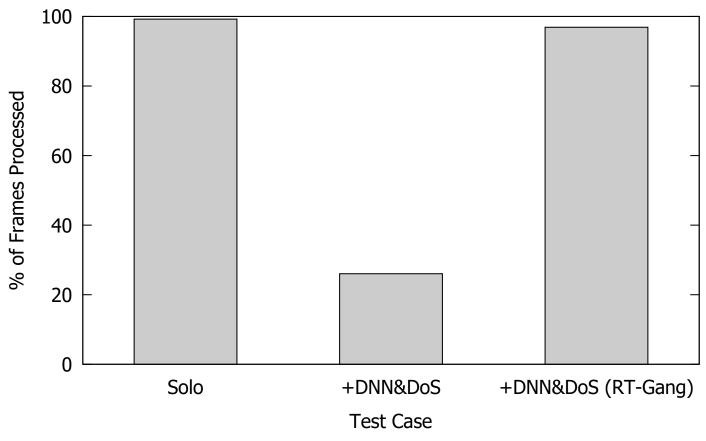


Figure 10 Percentage of input frames processed by OV^{2}SLAM with and without our extended version of RT-Gang enabled.

To determine RT-Gang's impact to best-effort tasks, we also measure the performance of the *BkPLLWrite(LLC)* attackers both with and without RT-Gang enabled. For this, we again run the *+DNN&DoS* and *+DNN&DoS (RT-Gang)* cases but measure the LLC bandwidth consumed by the DoS attacks in both scenarios.

Table 3 shows the best-effort bandwidth measured for each core. First, we find that the cores achieve the highest bandwidths when LLC throttling is not enabled. Then, when LLC

Core	No throttle	Dynamic Throttling (RT-Gang)	Static Throttling
0	2413	929	100
1	1304	827	100
2	5628	1155	100
3	156	118	100

Table 3 LLC bandwidth consumed by all best-effort *BkPLLWrite(LLC)* attackers. All values are in MB/s.

throttling is enabled in the RT-Gang framework (i.e. dynamic throttling), the best-effort tasks do achieve bandwidths greater than the 100 MB/s threshold as they are allowed full access to the LLC during slacks between real-time tasks. Surprisingly, we found that the core responsible for the DNN task, Core 3, consumed very little best-effort bandwidth. However, this is largely because the DNN task busy-waits on the CPU while waiting for the GPU kernel to finish, meaning there is very little real-time slack for that core. As such, the DNN task's core is unable to achieve higher best-effort LLC bandwidth. Overall, though, we do find our dynamic throttling approach to be a worthwhile solution. Compared to the case where static throttling is utilized (i.e. no core can consume more than 100 MB/s), we find that RT-Gang can allow for improved best-effort performance in a system while simultaneously protecting safety-critical real-time tasks.

Finally, to further test our extended version of RT-Gang, we run the OV^{2}SLAM task on the other four machine hall EuRoC scenarios, MH02-MH05. In particular, we run the same *Solo*, *+DNN&DoS*, and *+DNN&DoS (RT-Gang)* test cases performed previously.

Dataset	Solo	*+DNN&DoS*	*+DNN&DoS (RT-Gang)*
MH01	97.9	26.1	96.9
MH02	98.0	30.6	98.2
MH03	100	31.7	97.5
MH04	99.0	35.6	99.0
MH05	99.5	29.4	98.4

Table 4 Percentage of frames processed by OV^{2}SLAM for the MH02 - MH05 EuRoC datasets. All values are in %.

Dataset	Solo	*+DNN&DoS (RT-Gang)*		
	Median	Median	Min	Max
MH01	0.03	0.36	0.11	0.95
MH02	0.04	0.47	0.12	0.87
MH03	0.08	0.24	0.13	2.07
MH04	0.15	2.86	0.48	7.63
MH05	0.12	0.80	0.21	1.11

Table 5 OV^{2}SLAM median ATE on the Machine Hall scenarios from the EuRoC dataset. Note that we do not include the ATE for the *+DNN&DoS* test case as OV^{2}SLAM could not process the datasets in real-time, and only produced partial trajectories.

Table 4 shows the percentage of frames processed by OV^{2}SLAM in each test case. Much like the MH01 dataset, we find that OV^{2}SLAM is unable to process the majority of input frames in the *+DNN&DoS* case. When RT-Gang is enabled, though, we are again able to achieve real-time performance as OV^{2}SLAM processes >97% of input frames across all datasets.

Table 5 then shows the median ATE values for the trajectories generated by OV^{2}SLAM. Even with RT-Gang enabled, we still see notable performance drops in all test cases, mainly due to DRAM interference from the GPU. Addressing this is left for future work.

7 Related Works

Simultaneous Localization and Mapping (SLAM) algorithms are at the heart of many robotics applications, including ADAS and self-driving systems, as they are used to localize the position and pose of the ego-vehicle in connection with the surrounding environment. Due to the relatively cheaper cost of camera sensors, many efforts have been made towards solving the challenge of Visual SLAM. The PTAM algorithm was the first to introduce a multi-threaded Visual SLAM implementation [21] for improved real-time performance. It separated tracking and mapping into separate parallel threads on a dual-core processor, which allowed them to create more detailed 3D maps and achieve state-of-the-art accuracy. This type of multi-threaded approach has since been widely adopted in many Visual SLAM implementations. Notable examples of this include the LSD-SLAM algorithm [11], the many iterations of the ORB-SLAM algorithm [27, 28, 10], and the OV^{2}SLAM algorithm [13], which was used in this paper. These algorithms detect features of the camera images and track the features to locate their positions in the world. Other SLAM algorithms instead combine both camera and inertial sensors (e.g. IMU) to better account for visual artifacts such as low textures and motion blur. Better known as Visual-Inertial SLAM, or VI-SLAM for short, these algorithms allow for better robustness in situations where standard Visual SLAM algorithms may otherwise struggle. VI-SLAM approaches, though, tend to suffer from mid and long-term data association of generated 3D maps, meaning that the maps may not accurately reflect the surrounding environment if they are not updated frequently enough. To address this, the ORB-SLAM-VI algorithm used IMU preintegration to better allow for variable length data association [29]. The ORB-SLAM3 algorithm then extended upon this by using MAP estimation to better account for sensor uncertainties [10]. In this work, however, we focus on a Visual SLAM algorithm, OV^{2}SLAM, in our case study as it is recommended for the target AR HUD application.

Microarchitectural DoS attacks are software attacks specifically designed to induce a high-degree of resource contention. DoS attacks on several shared resources have been studied and evaluated. Moscibroda et al. proposed a DoS attack that targets a FR-FCFS scheduling algorithm [30] in shared DRAM controllers [26]. Keramidas et al. demonstrated DoS attacks targeting shared LLC space and, to address them, proposed a cache replacement policy that gave the attackers access to less of the LLC space [18]. Woo et al. investigated DoS attacks on bus bandwidth and shared cache space in a simulated environment [37]. Valsan et al. and Bechtel et al. showed that DoS attacks could target internal LLC hardware structures [7, 8, 36]. Based on [7], Iorga et al. presented a statistical approach for testing DoS attacks [17]. Li et al. applied machine learning to better optimize DoS attackers, resulting in WCET slowdowns >400X [23]. GPU-based DoS attacks have also been studied by researchers. Yandrofski et al, systematically studied shared resource contention on discrete Nvidia GPUs by generating various adversarial programs [39]. Bechtel et al., implemented DoS attacks targeted towards Intel iGPUs, as they also access the LLC.

Much effort has been devoted to address the problem of shared resource contention in multicore and many mitigation solutions have been proposed in the real-time systems research community, including both software and hardware-based mechanisms [25, 19, 40, 20, 31, 41, 35, 12, 38, 33]. Recently, the need for greater isolation in accessing shared hardware resources has also been recognized by major industry players. For example, Intel and ARM released their own tools, called RDT [16] and MPAM [4] respectively, that can be employed to provide better isolation in the LLC and DRAM. Despite these efforts, it is still not clear if these hardware mechanisms can provide sufficient isolation [43, 34, 6].

8 Conclusion

In this paper, we partly addressed the Industrial Challenge put forth by ARM in ECRTS 2022. We systematically analyzed the effect of shared resource contention to an augmented reality head-up display (AR-HUD) case-study application of the industrial challenge on a heterogeneous multicore platform. Using micro-architectural denial-of-service (DoS) attacks as aggressor tasks of the challenge, we showed that such aggressors can dramatically impact the latency and accuracy of the AR-HUD application, which could result in significant deviations of the estimated trajectories from the ground truth, despite the best effort to mitigate their influence by using cache partitioning and real-time scheduling of the AR-HUD application. We showed that selective LLC bandwidth throttling of the aggressor tasks is an effective mean to ensure real-time performance of the AR-HUD application without resorting to over-provisioning the system.

For future work, we plan to investigate mitigation strategies that can be used to prevent GPU-based interference from occurring. In particular, we plan to explore GPU bandwidth throttling as a way to limit GPU accesses to shared memory resources. We also plan to perform similar case studies on more capable platforms than the Jetson Nano, such as the Jetson Xavier or Jetson Orin lines of embedded platforms, to evaluate their susceptibility to shared resource contention.

References

1 Waqar Ali and Heechul Yun. RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems. In *RTAS*, 2019.

2 Matteo Andreozzi, Giacomo Gabrielli, Balaji Venu, and Giacomo Travaglini. Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm. In *ECRTS*, 2022.

3 SOAFEE. `https://www.soafee.io/`.

4 ARM. *Arm Architecture Reference Manual Supplement: Memory System Resource Partitioning and Monitoring (MPAM), DDI:0598B.b*, 2020.

5 Michael Bechtel and Heechul Yun. Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems. *Transactions on Computers*, 2021.

6 Michael Bechtel and Heechul Yun. Cache Bank-Aware Denial-of-Service Attacks on Multicore ARM Processors. In *RTAS*, 2023.

7 Michael G Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *RTAS*, 2019.

8 Michael Garrett Bechtel, Elise McEllhiney, Minje Kim, and Heechul Yun. DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car. In *RTCSA*, 2018.

9 Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The EuRoC Micro Aerial Vehicle Datasets. *The International Journal of Robotics Research*, 2016.

10 Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 2021.

11 Jakob Engel, Thomas Schöps, and Daniel Cremers. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *ECCV*, 2014.

12 Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *ECRTS*, 2018.

13 Maxime Ferrera, Alexandre Eudes, Julien Moras, Martial Sanfourche, and Guy Le Besnerais. OV^{2}SLAM: A Fully Online and Versatile Visual SLAM for Real-Time Applications. *IEEE robotics and automation letters*, 2021.

14 HopeNet-Lite. `https://github.com/OverEuro/deep-head-pose-lite`.

15 rosbag2. `https://github.com/ros2/rosbag2`.

16 Intel. *Intel 64 and IA-32 Architectures Software Developer Manuals, Vol 3b*, May 2020.

17 Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F Donaldson. Slow and Steady: Measuring and Tuning Multicore Interference. In *RTAS*, 2020.

18 Georgios Keramidas, Pavlos Petoumenos, Stefanos Kaxiras, Alexandros Antonopoulos, and Dimitrios Serpanos. Preventing denial-of-service attacks in shared cmp caches. In *SAMOS*, 2006.

19 H. Kim, A. Kandhalu, and R. Rajkumar. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *ECRTS*, 2013.

20 Namhoon Kim, Bryan C Ward, Micaiah Chisholm, James H Anderson, and F Donelson Smith. Attacking the One-Out-of-M Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. *Real-Time Systems*, 2017.

21 Georg Klein and David Murray. Parallel Tracking and Mapping for Small AR Workspaces. In *ISMAR*, 2007.

22 Ao Li, Han Liu, Jinwen Wang, and Ning Zhang. From Timing Variations to Performance Degradation: Understanding and Mitigating the Impact of Software Execution Timing in SLAM. In *IROS*, 2022.

23 Ao Li, Marion Sudvarg, Han Liu, Zhiyuan Yu, Chris Gill, and Ning Zhang. PolyRhythm: Adaptive Tuning of a Multi-Channel Attack Template for Timing Interference. In *RTSS*, 2022.

24 Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet V2: Practical Guidelines for Efficient CNN Architecture Design. In *ECCV*, 2018.

25 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-Time Cache Management Framework for Multi-core Architectures. In *RTAS*, 2013.

26 Thomas Moscibroda and Onur Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security Symposium*, 2007.

27 Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE transactions on robotics*, 2015.

28 Raul Mur-Artal and Juan D Tardós. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE transactions on robotics*, 2017.

29 Raúl Mur-Artal and Juan D Tardós. Visual-Inertial Monocular SLAM Sith Map Reuse. *IEEE Robotics and Automation Letters*, 2017.

30 S. Rixner, W. J Dally, U. J Kapasi, P. Mattson, and J. Owens. Memory Access Scheduling. In *ACM SIGARCH Computer Architecture News*, 2000.

31 Shahin Roozkhosh and Renato Mancuso. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *RTAS*, 2020.

32 Nataniel Ruiz, Eunji Chong, and James M. Rehg. Fine-Grained Head Pose Estimation Without Keypoints. In *CVPR*, 2018.

33 Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In *RTAS*, 2022.

34 Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A Closer Look at Intel Resource Director Technology (RDT). In *RTNS*, pages 127–139, 2022.

35 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-WarP: a System-wide Framework for Memory Bandwidth Profiling and Management. In *RTSS*, 2020.

36 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *RTAS*, 2016.

37 D Hyuk Woo and HH Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *CMP-MSI*, 2007.

38 Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. Holistic Resource Allocation for Multicore Real-Time Systems. In *RTAS*, 2019.

39 Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H Anderson, and FD Smith. Making Powerful Enemies on NVIDIA GPUs. In *RTSS*, 2022.

40 Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a Dynamic Cache Partitioning System Using Page Coloring. In *PACT*, 2014.

41 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *RTAS*, 2013.

42 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *RTAS*, 2014.

43 Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing arm's mpam from the perspective of time predictability. *IEEE Transactions on Computers*, 72(1):168–182, 2022.