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

Michael Bechtel, Heechul Yun

Abstract—In this paper, we present a solution to the industrial challenge put forth by ARM in 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 Microarchitectural Denial-of-Service (DoS) attacks as aggressor workloads of the challenge and show that they can dramatically impact the latency and accuracy of the AR-HUD application. This results in significant deviations of the estimated trajectories from known ground truths, despite our best effort to mitigate their influence by using cache partitioning and real-time scheduling of the AR-HUD application. To address the challenge, we propose RT-Gang++, a partitioned real-time gang scheduling framework with last-level cache (LLC) and integrated GPU bandwidth throttling capabilities. By applying RT-Gang++, we are able to achieve desired level of performance of the AR-HUD application even in the presence of fully loaded aggressor tasks.

Index Terms—Industrial Challenge, Real Time, SLAM, Microarchitectural DoS Attack

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 on shared microarchitectural resources, such as shared cache and main memory, between the computing elements in such a platform remains a significant challenge because it can impact the execution timings of critical real-time tasks and thus jeopardize the safety of the CPS. Moreover, shared resource contention can also be intentionally induced by malicious actors with the goal of compromising the performance and safety of CPS. Such adversaries are known as microarchitectural Denial-of-Service (DoS) attacks [9], and are especially problematic for high-performance CPS that need to run multiple concurrent applications simultaneously on a single multicore platform. Given the recent trends towards connected CPS, as can be seen in ARM's SOAFEE initiative [4] 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 2022 to address the problem of shared resource contention [3]. 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 [15] and a DNN-based driver head pose estimation task [16]. 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 to it as the DNN task) may utilize the GPU. The application represents a computationally intensive mixed-criticality real-time 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.

In this paper, we first 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 (the SLAM algorithm and head pose detection) in the AR-HUD case-study on a representative heterogeneous system-on-chip (SoC) processor and achieve required real-time performance (i.e., meeting the deadlines)? (2) Does shared resource contention between the two AR-HUD tasks impact the accuracy of the obtained position/trajectory estimates of the SLAM task? (3) Can we guarantee a desired level of performance, in terms of both accuracy and latency, 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 GPU. Our findings are as follows: We are able to configure the

1. *Dr. Bechtel is currently with Garmin. This work was conducted while he was with the University of Kansas. E-mail: mgbechte@gmail.com*

2. *Dr. Yun is with the University of Kansas and is the corresponding author of this manuscript. E-mail: heechul.yun@ku.edu*

AR-HUD application such that it can process incoming image frames in real-time at 20Hz. However, we find that contention between the two real-time tasks does significantly impact the accuracy of the SLAM task, 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 at 20Hz. In addition, we find that cache bank-aware DoS attack [7] is especially effective in impacting the accuracy and real-time performance of the AR-HUD application. Concretely, when the cache bank-aware DoS attack tasks are co-scheduled as best-effort (non-RT) tasks together with the real-time tasks of the AR-HUD application to fully load the system, the SLAM task fails to even generate the trajectory as it has to drop most of the incoming image frames due to increased latency caused by contention.

To address the challenge, we propose RT-Gang++, a partitioned real-time gang scheduling framework with iGPU and last level cache (LLC) bandwidth throttling capabilities. RT-Gang++ is based on [2] but extends its capabilities as follows: (1) add support for partitioned gang-scheduling to allow for multiple real-time gangs of different priorities to execute concurrently; (2) add support for LLC and iGPU bandwidth throttling to protect against contention on those shared resources. These additional capabilities are crucial to address the ARM industrial challenge problem.

By employing RT-Gang++, we are able to safely consolidate the AR-HUD application on the Jetson Nano platform, even in the presence of malicious DoS attacks, and achieve desired real-time performance and accuracy. In addition, we also ported RT-Gang++ on a Raspberry Pi 4 platform, and evaluate its effectiveness. For reproducible dissemination, we release the AR-HUD application setup, including our ROS2 port of OV^{2}SLAM, evaluation scripts, and RT-Gang++, as open-source [1].

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. Section 7 presents the results. We review related work in Section 8 and conclude in Section 9.

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

Addressing the impacts of shared resource contention is of critical importance for 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 physical 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. The 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 [15] is suggested as part of the AR-HUD case study [3]. OV^{2}SLAM is a Visual SLAM algorithm that is geared toward 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, but does not create a keyframe for every given frame. 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.
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 [3].

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.

1. https://github.com/CSL-KU/ArmArHudChallenge

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 [16], 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 [3]. These aggressor workloads may be co-scheduled alongside the AR-HUD application and may contend for shared resources.

For our aggressor workloads, we employ microarchitectural denial-of-service (DoS) attacks that have been described in literature [38], [9], [7]. 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 [38]. 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 [38]. 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 [7]. Much like memory-aware attacks [6], these attacks are 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.

Hardware Platform: For the hardware platform, we use an Nvidia Jetson Nano platform, which equips a quad-core Cortex-A57 cores with each core having its own private L1 instruction and data caches and all cores sharing a global L2 cache. Table 1 shows the basic characteristics of the platform.

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.

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 [3], 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 is at a fixed rate of 20Hz. The *Mapping* and *State Optimization* threads are invoked conditionally when the *Front End* thread generates a new key frame (See Section 2 for details.)

As the SLAM task is *real-time critical* [3], 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 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.

As for input data for the SLAM task, we use the five Machine Hall (MH) scenarios of the EuRoC dataset [10]. Note that the EuRoC dataset includes visual-inertial data of a micro aerial vehicle (MAV), which includes stereo images, IMU measurements, and accurate motion and structure ground-truth data. In the MH scenarios, the data were collected while traversing through an indoor environment populated with various machinery with a varying degree of complexities, with the MH01 scenario being the easiest one for SLAM and the MH05 scenario being the hardest one.

The image frames from the dataset are fed to the SLAM task through an instance of `rosbag2` [17], a tool that plays back datasets recorded in ROS bag files, which was running on Core 2 with a real-time priority of 2 such that it is not delayed by any best-effort tasks. Note that all MH datasets playback input data at a frequency of 20Hz. The observed CPU utilization of `rosbag2` is about ∼5-10% of the Core 2.

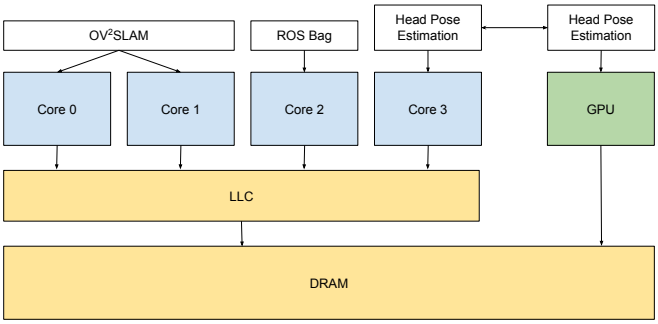


Fig. 1: Tasks to core assignments of the AR-HUD case-study on Jetson Nano.

For the head pose estimation task, we use the recommended HopeNet-Lite DNN model [16], which is a lightweight version of the original HopeNet model [33]. Note that the Jetson Nano mainly processes the HopeNet-Lite model on the GPU but a single CPU core (Core 3) is also used to launch the GPU kernel and monitor its progress. On the Jetson Nano we configure the DNN 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* [3]. 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. In addition, Table 2 shows the task/thread/core mapping and real-time scheduling parameters of all real-time tasks in the AR-HUD case study.

Operating System Setup: For the Jetson Nano's 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 by 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.

Task	Thread	Core(s)	RT Priority	Rate (Hz)
OV^{2}SLAM	Front-End	0,1	2	20
	Mapping			-
	State Optimization			-
ROS Bag	-	2	2	20
Head Pose Est.	-	3,GPU	1	20

TABLE 2: Real-time tasks/threads/core mapping and scheduling parameters in the AR-HUD case study on the Jetson Nano. Note that all real-time tasks are scheduled using the `SCHED_FIFO` real-time scheduler and a bigger priority value indicates a higher real-time priority.

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 do 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 OV^{2}SLAM task using the MH01 scenario in the EuRoC dataset [10]. Once finished, we calculate the algorithm's *Absolute Trajectory Error (ATE)* relative to the known ground-truth trajectory [10]. 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 (scheduled using the CFS scheduler) while run the OV^{2}SLAM task as a real-time task (using the `SCHED_FIFO`). Because Linux strictly prioritizes real-time tasks over best-effort ones, the DoS attack tasks can only be executed on cores which are not executing any of the RT tasks. In other words, whenever the threads of 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. This separation minimizes any negative effect of co-scheduling as the DoS attack tasks cannot evict cache-lines

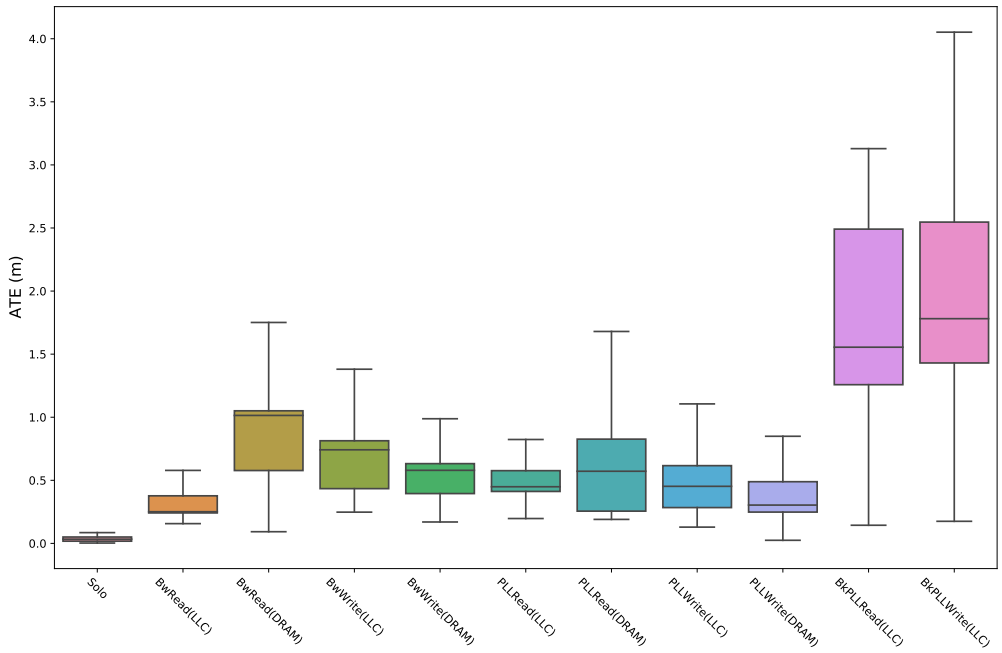


Fig. 2: Impact of DoS attacks on the Absolute Trajectory Error (ATE) of the OV^{2}SLAM generated trajectory.

of the SLAM task. Therefore, any observed delays are not attributable to CPU scheduling or cache space contention.

Figure 2 shows a boxplot of the OV^{2}SLAM ATE collected over the entire duration of the MH01 dataset, both alone (Solo) and alongside each of the tested DoS attacks (the rest in the X-axis).

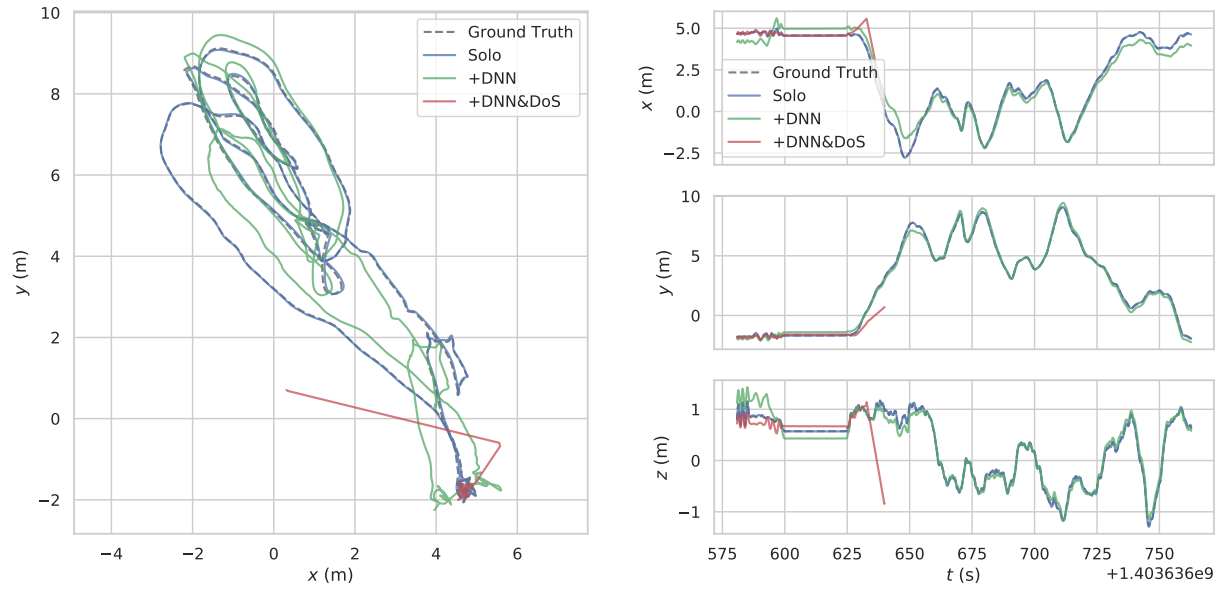
Firstly, note that all of the tested DoS attacks cause significant negative impacts to the tracking performance, resulting in ATE increase of up to 4.0 meters. This occurs despite the fact that the DoS attacks cannot preempt the SLAM task or evict its cache-lines. This is because there are many other shared hardware resources that can impact execution timing in modern multicore. These shared hardware resources include DRAM bandwidth [43], [6], DRAM bank [42], cache internal buffers/queues [38], [9], and cache bank [7]. The DoS attacks in the X-axis are designed to induce maximum contention in those shared resources. Note that, for this scenario, ATEs of $\sim$0.3 or more indicate significant deviations from the ground truth, which could potentially cause failure (e.g., a crash) in the real-world [23].

In particular, we observe that the cache bank-aware DoS attacks recently proposed in [7], denoted as BkPLLRead(LLC) and BkPLLWrite(LLC) for read and write, respectively, are particularly effective in influencing ATE. Specifically, the *BkPLLRead(LLC)* attack increased the median ATE to over 1.7 (49X increase over solo), and the *BkPLLWrite(LLC)* attack increased it to over 1.9 (55X increase). In simpler terms, these attacks caused OV^{2}SLAM's detected trajectory to deviate from the ground truth trajectory by a median value of two meters, and more than four meters in the worst case. Both cache bank-aware DoS attacks are specially designed to generate many concurrent accesses to a specific LLC cache bank, causing contention on the bank. The shared L2 cache of Cortex-A57 consists of two tag banks, each composed of four data banks that can be accessed in parallel [7]. By directing concurrent accesses to a single data bank, the SLAM task's access to the cache bank is delayed, subsequently delaying the execution of the SLAM task.

5.2 Impact of Co-scheduling HopeNet-Lite on 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: *Solo*, *+DNN*, and *+DNN&DoS*. In *Solo*, the OV^{2}SLAM runs alone; In *+DNN*, the HopeNet-Lite DNN is co-scheduled with OV^{2}SLAM; In *+DNN&DoS*, 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. This implies that the DoS attackers cannot preempt OV^{2}SLAM or HopeNet-Lite. Based on the findings in Section 5.1, we opt for the *BkPLLWrite(LLC)* attack as the aggressor workloads as it is the most effective at degrading the performance of the SLAM task.

Figure 3 shows the ground-truth and the three generated trajectories. First, in *Solo*, the generated trajectory almost completely overlaps with the ground-truth, indicating OV^{2}SLAM achieves good accuracy, with a median ATE of 0.03m, meaning the observed trajectory error is only about 3 centimeter. In *+DNN*, however, the addition of the HopeNet-Lite DNN task significantly impacts the accuracy of OV^{2}SLAM with the median ATE increasing to over 0.8m (approximately 27X increase over solo). This is because the OV^{2}SLAM and the HopeNet-Lite DNN task compete for shared hardware resources, particularly the shared DRAM, which is shared between the CPU (executing the SLAM) and the integrated GPU (executing the DNN). Lastly, in *+DNN&DoS*, when HopeNet-Lite is combined with the DoS attacks, OV^{2}SLAM suffers a drastic performance degradation, leading to a complete failure in generating a full



(a) Trajectory in XY plane

(b) X, Y and Z positions over time

Fig. 3: OV^{2}SLAM trajectory when run alongside both *BkPLLWrite(LLC)* DoS attackers and a GPU-based DNN application.

trajectory. This failure is due to OV^{2}SLAM's inability to keep up with the input data, resulting in dropping a majority of the input camera frames. Our analysis shows that only about 26% of the input image frames were processed in *+DNN&DoS*, compared to over 97% in either the *Solo* or *+DNN* cases. Once again, contention on shared hardware resources among the co-scheduled tasks, particularly on the shared DRAM (caused by the DNN task) and the LLC bank (caused by the DoS attacks), contributes to this performance degradation.

In summary, co-scheduling HopeNet-Lite DNN and/or BkPLLWrite(LLC) DoS attacks significantly affects the accuracy of OV^{2}SLAM due to contention on shared hardware resources, despite the CPU and cache space being partitioned to protect the OV^{2}SLAM task.

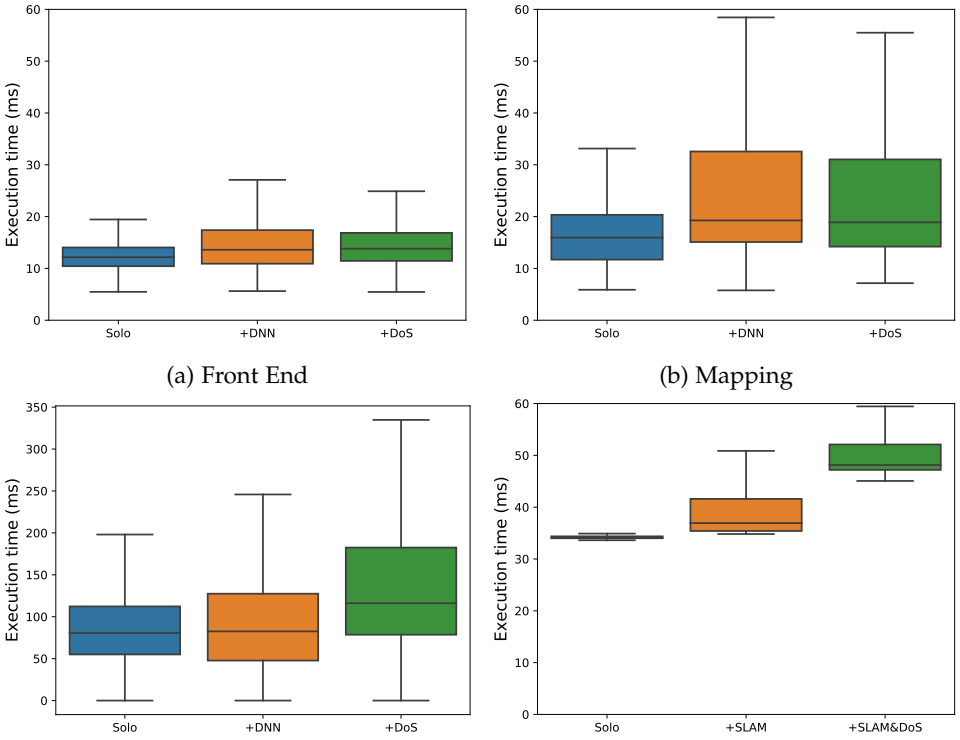
5.3 Runtime Analysis of OV^{2}SLAM and HopeNet-Lite

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 OV^{2}SLAM (visual SLAM), we measure and record the execution times of each thread of the task when they execute their main computational loop (e.g. *Front End* receives a new input frames, *Mapping* receives a new keyframe, etc.). We re-run the experiments on the following three scenarios: *Solo* when the SLAM task and its three threads run alone; *+DoS* when the co-schedule the *BkPLLWrite(LLC)* DoS attack tasks alongside with the SLAM; and *+DNN* when we co-schedule the HopeNet-Lite task.

For HopeNet-Lite (head pose estimation), 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. We re-run the experiments on the following three scenarios: *Solo* when it runs alone, *+SLAM* when run with the OV^{2}SLAM task, and *+SLAM&DoS* when run with both the OV^{2}SLAM task and *BkPLLWrite(LLC)* DoS attack.

Figure 4 shows the execution time distributions for all real-time tasks (OV^{2}SLAM and HopeNet-Lite DNN) and their threads in each of the tested scenarios. First, for the OV^{2}SLAM task, we observe execution time increases in all three threads when co-runners (the HopeNet-Lite DNN or DoS attacks) are present. In case of the *Front End* thread, inset (a), the execution time increases due to either type of the co-runners are relatively small. In case of the *Mapping* thread, inset (b), both the DNN and the DoS attackers significantly increase the mapping thread's execution time. In case of the *State Optimization* thread, inset (c), on the other hand, the DoS attackers are more effective than the DNN co-runner in increasing its execution time. Recall that the DNN and the DoS tasks cause contention on different shared hardware resources, namely DRAM (bandwidth) and LLC (bank), respectively. As such, we can infer that (1) the Front End thread is not very sensitive to either of the shared resources, (2) the Mapping thread is sensitive to both DRAM and LLC, and (3) the State Optimization thread is more sensitive to the LLC. These execution time increases, especially in Mapping and State Optimization threads, due to contention result in the ATE increases we observe when the DNN or the DoS attack tasks are present. Note that the observed ATE loss is much higher when the DoS attackers are present, compared to when only the DNN co-runner is present (55X vs. 27X ATE increase over Solo), suggesting the importance of State Optimization in the overall accuracy of the OV^{2}SLAM algorithm, which is consistent with prior findings [23].



(a) Front End (b) Mapping

(c) State Optimization (d) Head Pose Estimation

Fig. 4: Execution time distributions of all real-time tasks of the AR-HUD: (a)-(c) OV^{2}SLAM; (d) HopeNet-Lite

Second, inset (d) shows the execution time distributions of the HopeNet-Lite DNN task. Note that when it runs together with the SLAM (on different cores/GPU), its average inference time increase from ∼34 ms (solo) to ∼37 ms (+SLAM). When the DoS attacks are then added (+SLAM&DoS), however, the average inference time increases again to ∼48 ms. From these, we can infer that the DNN task itself is sensitive to both the SLAM task—especially with its Mapping thread, which may be contending DRAM bandwidth with the DNN—and the DoS attack task, which causes contention on a specific LLC bank.

6 Mitigating Shared Resource Contention

In this section, we present a mitigation solution to protect the real-time AR-HUD application in the presence of aggressor tasks (i.e., DoS attacks).

6.1 RT-Gang++

In this work, we leverage the RT-Gang scheduling framework, which is a real-time gang scheduler, implemented as an extension to the `SCHED_FIFO` real-time scheduler in the Linux kernel [2]. 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.

When we tried to apply RT-Gang to mitigate the shared resource contention problem in the AR-HUD application of the ARM industrial challenge, we encountered the following challenges. First, the original RT-Gang supports only a single gang task at a time. When we schedule the real-time tasks as separate gang tasks in the original RT-Gang, they cannot meet the necessary real-time requirements because they are not fully parallelized, not taking advantage of all available cores and the GPU. Second, even if we group them together to form a "virtual gang" task [1] to improve resource utilization, RT-Gang does not offer any contention mitigation mechanisms between the real-time tasks within the virtual gang. This means that there is no way to minimize negative performance impact on a higher priority OV^{2}SLAM task due to co-scheduling the lower priority DNN real-time task, which runs on the iGPU. Lastly, when we deploy the cache bank-aware DoS attack [7] as the best-effort aggressor tasks, RT-Gang's memory bandwidth throttling capability becomes ineffective in protecting the real-time tasks because the aggressors do not consume any memory bandwidth as they target LLC bank contention.

To address these challenges, we make three *extensions* to the vanilla RT-Gang: (1) partitioned real-time gang scheduling capability; (2) iGPU bandwidth throttling; (3) LLC bandwidth throttling. We call the resulting system *RT-Gang++*.

6.2 Partitioned Gang Scheduling

One major feature of the baseline RT-Gang is that it allows only one real-time gang task at a time across all the cores in a multicore CPU. While it does prevent shared resource contention between RT tasks by design, which is desirable for predictability, it is also a limitation in terms of scalability because not all tasks can benefit from a large number of cores and the number of cores in CPUs keeps increasing. While RT-Gang somewhat mitigates the problem by supporting so called "virtual gangs", which is a collection of multiple real-time tasks with the same period and the same priority that collectively acts like a single gang task, it cannot be used when either the priority or the period of any RT task differs from the rest. Moreover, many modern multicore CPUs are often composed of multiple clusters, each of which may have different sets of computing and memory resources that are not shared with the rest. For example, many ARM multicore CPUs incorporate the big.LITTLE architecture where one cluster is composed of powerful "big" cores and the other cluster is composed of efficient "little" cores. Not only do the clusters have different types of CPU cores, but they also often have cluster-private shared resources that are not shared across the clusters, which reduces the need to strictly adhere to the one-gang-task-at-a-time policy of RT-Gang.

In RT-Gang++, we support multiple *partitions* where each partition is composed of a statically determined set of cores. The one-gang-task-at-a-time is then applied to each partition rather then being applied globally. This means that multiple gang tasks, each with different priority and/or period, can run simultaneously as long as they are assigned to different partitions. For our AR-HUD case-study in Table 2, we assign OV^{2}SLAM and EuRoC playback tasks to form a virtual gang task and assign it into a gang partition, which is comprised of core 0, 1, and 2, while assigning the DNN task (head pose estimation) on another gang partition, which is composed of the core 3 and the iGPU. In this way, the system can run two active gang tasks with different priority and period simultaneously.

6.3 iGPU Memory Bandwidth Throttling

When multiple real-time tasks run together, however, they inevitably contend on the shared resources. As observed in Section 5.2, co-scheduling the DNN task in particular has detrimental effect to the more critical OV^{2}SLAM task's accuracy. The baseline RT-Gang, unfortunately, does not provide any means to address the contention between the two co-scheduled RT tasks. In this work, we leverage hardware-level GPU bandwidth throttling capability of the platform we used for evaluation. Specifically, the Tegra X1 SoC of the Jetson Nano platform supports a number of QoS features at its memory controller, one of which is hardware-level throttling of subsets of hardware components that access the memory controller [30]. The throttling feature of the memory controller provides 32 programmable throttling levels that can be applied to throttle the integrated GPU, which we used to throttle the DNN task whose memory access from the GPU impacts the performance of the OV^{2}SLAM task.

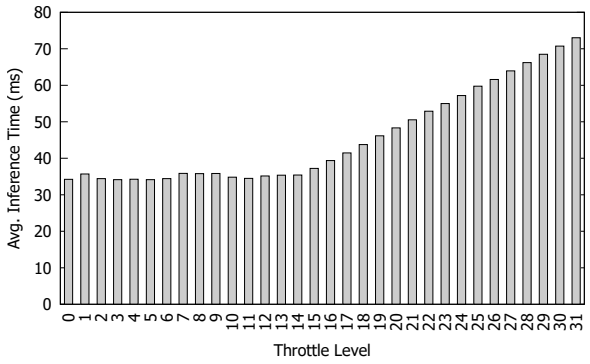


Fig. 5: Impact of GPU throttling on HopeNet-Lite DNN.

Figure 5 shows the average HopeNet-Lite inference latencies under each of the 32 throttling levels when running alone in isolation. Note that GPU throttling does not impact the performance of the DNN task until the throttling level reaches around 15, indicating that sufficient bandwidth is provided until that point. After that, more aggressive throttling does impact the performance of the DNN. When the throttling level is 31, which is the maximum, the average inference latency is increased to $\sim$73 ms, which is about twice longer than without throttling. Note that there is a tradeoff between the accuracy of the SLAM task and the latency of the DNN task as more aggressive throttling of the GPU, which is used by the latter, will be helpful to achieve higher accuracy (lower ATE) for the SLAM task but it will increase the latency of the DNN task. As such, finding a "sweet spot" for the target application is necessary. For our study, we experimentally chose the level 20 as it was the maximum throttling level that still can provide 20Hz real-time performance for the DNN inference task. However, one can choose a more aggressive throttling level (e.g., the level 31) if achieving the highest accuracy of the SLAM task is more important than processing the DNN task at 20Hz. In our testing, using the GPU throttling level 31 allows the SLAM task to achieve near perfect isolation but at the cost of doubling the latency of the DNN task.

6.4 LLC Bandwidth Throttling

For our case-study application, both the OV^{2}SLAM and DNN real-time tasks must be protected from the interference of co-scheduled aggressor tasks, which are scheduled in a best-effort manner (i.e., scheduled on any cores that do not execute RT tasks). As discussed in Section 5.1, when DoS attackers are used as the aggressor tasks, the performance of the OV^{2}SLAM algorithm is significantly reduced even though the DoS attackers cannot preempt the SLAM task due to shared resource contention. In particular, we find that cache bank-aware DoS attack [7] is particularly effective in negatively impacting accuracy of the SLAM task. Unfortunately, however, the baseline RT-Gang's memory bandwidth throttling capability does not provide any protection against the cache-bank DoS attack, as it generates LLC cache hits and does not consume any memory bandwidth.

In RT-Gang++, we add support for LLC bandwidth throttling capability by utilizing the L1-D cache miss performance counter (`L1D_CACHE_REFILL`) of the CPU cores

to track and throttle LLC (L2) bandwidth used by the best-effort DoS attacker tasks. Note that the throttling implementation is based on MemGuard [43] and we use an experimentally determined LLC bandwidth threshold of 100 MB/s when LLC throttling is enabled and the regulation interval is 1ms.

7 Evaluation Results

In this section, we evaluate the performance of RT-Gang++ on two popular embedded multicore platforms: Jetson Nano and Raspberry Pi 4.

7.1 Jetson Nano

The basic experiment setup is the same as described in Section 4: that is, we execute the SLAM task on Cores 0 and 1, the EuRoC dataset playback task on Core 2, and the DNN task on Core 3 and the iGPU.

For evaluation, we repeat the experiment in Section 5.2 but with using RT-Gang++. Recall that in this experiment, we co-schedule BkPLLWrite(LLC) DoS attackers as aggressors on all CPU cores to incur contention on the shared hardware resources, specifically on a single LLC bank. As such, we want to know how well RT-Gang++ can protect the performance of the SLAM and the DNN task in the presence of the DoS attackers.

7.1.1 Results

Figure 6 shows the trajectories generated by the OV^{2}SLAM task, and the X, Y and Z positions of the trajectory over time. Note that *Co-run* denotes the baseline configuration without using RT-Gang++, while *RT-Gang++* denotes the same configuration with RT-Gang++ enabled. As observed earlier, in *Co-run*, the OV^{2}SLAM fails to generate valid trajectory due to contention. In *RT-Gang++*, however, the generated trajectory of the OV^{2}SLAM is valid and much closer to that of the *Solo* and ground-truth, showing the effectiveness of the RT-Gang++ in protecting the performance of the SLAM task.

Configuration	Avg. inference time (ms)
Solo	34.23
Co-run	36.32
RT-Gang++	48.69

TABLE 3: Average inference latency of the DNN task

Table 3 shows the average inference times achieved by the HopteNet-Lite DNN task on three different test configurations. In *Solo*, the DNN task runs alone in isolation and achieves an average inference latency of ∼34ms per image frame. In *Co-run*, the SLAM and the DoS attackers are co-scheduled with the DNN, which results in a slight increase, going from ∼34 to ∼36 ms on average. In *RT-Gang++*, on the other hand, the average latency of the DNN task is increased to ∼49 ms. This is because iGPU throttling limits the iGPU's DRAM bandwidth usage, which makes the DNN task runs slower, which in turn help protect the performance of the SLAM task, at the cost of the DNN. Nevertheless, it is important to note that the DNN task is still able to achieve the desired 20Hz rate (per Table 2).

Core	Solo	Co-run	RT-Gang++
0	8241	2413	956
1	8470	1304	792
2	7643	5628	1211
3	7496	156	119

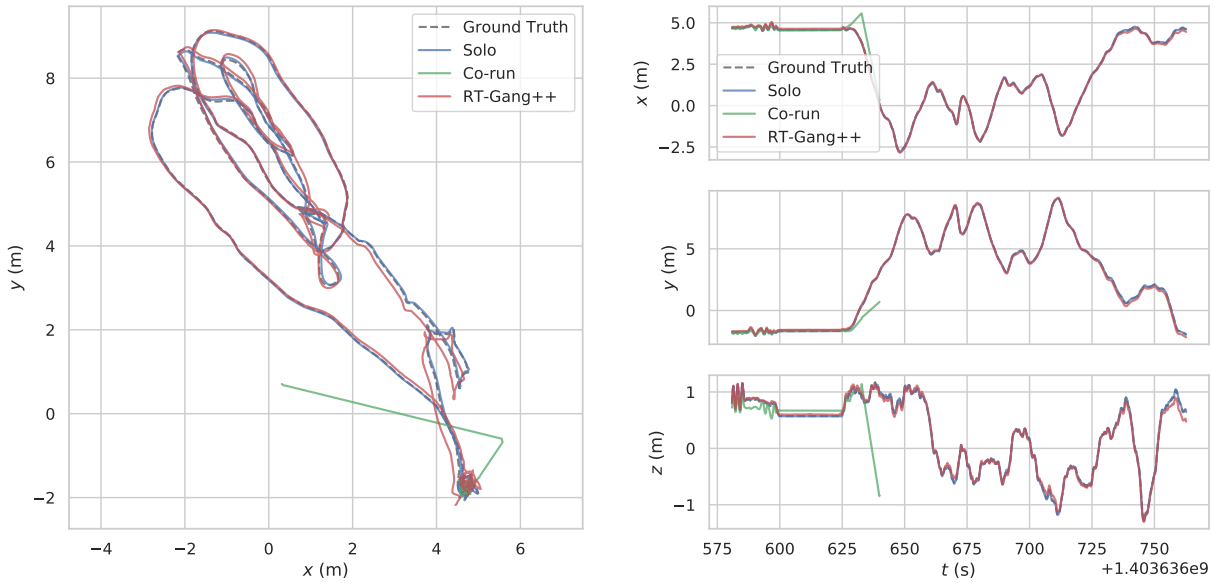
TABLE 4: Average LLC bandwidth (in MB/s) consumed by all best-effort *BkPLLWrite(LLC)* attackers.

Table 4 shows the average LLC bandwidth consumed by the best-effort DoS attackers on each core first alone in isolation, in *Solo*, together with both SLAM and DNN real-time tasks without and with RT-Gang++, in *Co-run* and *RT-Gang++*, respectively. Recall that RT-Gang++ throttles LLC bandwidth of best-effort tasks to limit the LLC bank contention. As a result, the average LLC bandwidth numbers of *RT-Gang++* is much lower than that of *Co-run* or *Solo*, which is expected. Nevertheless, it is important to note that these best-effort DoS attackers are still able to use a significantly higher LLC bandwidth that the set threshold of 100MB/s (which was chosen experimentally per Section 6.4). This is because, in RT-Gang++, the LLC bandwidth throttling is dynamically enabled only when there are currently scheduled real-time tasks in any of the CPU cores in the platform. When there are no active real-time tasks, which happen often as they can finish earlier than the deadlines, LLC bandwidth throttling is automatically disabled, which allows the best-effort DoS attackers to fully utilize the full LLC bandwidth without throttling, until any of the real-time tasks become active again. This is

Dataset	Solo	RT-Gang++ (No iGPU throttling)	RT-Gang++
MH01	0.03	0.36	0.11
MH02	0.04	0.47	0.07
MH03	0.08	0.24	0.20
MH04	0.15	2.86	0.29
MH05	0.12	0.80	0.23

TABLE 5: OV^{2}SLAM median ATE (in meters) on the Machine Hall scenarios from the EuRoC dataset.

Table 5 then shows the median ATE values of the OV^{2}SLAM generated trajectories on all five machine hall scenarios of the EuRoC dataset. Note that *Co-run* is not included as OV^{2}SLAM fails to generate full trajectories due to contention. Even with RT-Gang++, we still see notable increases in reported ATE values when the iGPU bandwidth throttling is not enabled. On the other hand, when both LLC and iGPU bandwidth throttling are enabled in *RT-Gang++*, we observe significant improvements in ATE. The degree of improvements differs depending on the scenarios. For example, the MH04 scenario, which is among the hardest ones, is particularly sensitive to the iGPU throttling as the SLAM's ATE is whopping 2.86 meters without iGPU throttling, while it is only 0.29 meters with iGPU throttling. The results show that both iGPU throttling and LLC bandwidth throttling are important to protect the performance of the SLAM task. Note that an ATE value less than 0.3 meters is required for these scenarios [23], which we are able to satisfy in all cases with RT-Gang++ fully enabled.



(a) Trajectory in X-Y plane

(b) X, Y, and Z positions over time.

Fig. 6: Impact of RT-Gang++ on OV^{2}SLAM performance on the Jetson Nano.

7.2 Raspberry Pi 4

To demonstrate the generality of RT-Gang++, we additionally evaluate it on a Raspberry Pi 4 platform.

7.2.1 Hardware and Software Setup

The Raspberry Pi 4 features a Broadcom BCM2711 SoC, which includes a quad-core ARM Cortex-A72 CPU with a 1MB shared L2 cache, a VideoCore 6 GPU, and 4GB LPDDR4 SDRAM. Note that the Raspberry Pi 4's Cortex-A72 CPU cores are more advanced and powerful than the Jetson Nano's Cortex-A57 cores. The Pi 4's VideoCore 6 GPU, though, is weaker than the Nano's 128-core Maxwell GPU and lacks software framework support for GPU offloading. Table 6 shows the hardware specification for the Raspberry Pi 4.

Platform	Raspberry Pi 4
SoC	BCM2711
CPU	4x Cortex-A72 @ 1.5GHz
GPU	VideoCore 6 (not used)
Shared LLC (L2)	1MB (16-way)
Memory (Peak B/W)	4GB LPDDR4 (25.6 GB/s)

TABLE 6: Raspberry Pi 4 hardware specifications.

The basic application setup is the same as that of the Jetson Nano in Table 2, except that HopeNet-Lite DNN task runs entirely on the CPU due to the lack of DNN software support for the Pi 4's GPU.

For the software, the Pi 4 runs Raspberry Pi OS with Linux kernel 5.15 and is also patched with PALLOC [42] to support LLC space partitioning, as was the case for the Jetson Nano platform. We use the same 2/2 split LLC partitioning setup as we did on the Jetson Nano, although the size of each cache partition is halved as the Pi 4's L2 cache size is a half of Nano's.

Lastly, we ported RT-Gang++ on the Pi 4's Linux kernel, including partitioned gang scheduling and LLC bandwidth throttling capabilities. However, the iGPU throttling is not implemented because the iGPU is not used as noted earlier.

7.2.2 Results

First, we repeat the experiment in Section 5.1 to understand the effect of various DoS attacks on the Pi 4 platform.

Figure 7 shows the results. Similar to the results on Jetson Nano in Figure 2, all DoS attacks increase ATE scores in the SLAM generated trajectories due to increased contention. Note also that BkPLLWrite(LLC) is again the most effective DoS attack, same as the Jetson Nano platform, although its median ATE increase of 1.3 is somewhat smaller than the median ATE of 1.9 we observe on the Jetson Nano platform. This suggests that Pi 4's Cortex-A72 CPU cores are also highly susceptible to LLC bank contention as in Nano's Cortex-A57. As such, we again use the BkPLLWrite(LLC) as the DoS attacker task in the subsequent experiments.

Next, we repeat the experiment in Section 5.2 to evaluate the performance of RT-Gang++ in protecting the performance of the OV^{2}SLAM in the presence of the DoS attacks and the HopeNet-Lite DNN task.

Figure 8 shows the OV^{2}SLAM generated trajectories and XYZ positions over time with and without RT-Gang++ on the Pi 4 platform. Note first that, like the results on the Jetson Nano platform (Figure 6), we find that contention from the co-scheduled DoS attacks and the DNN task do impact performance of the SLAM. Unlike on Jetson Nano, however, the SLAM task is able to generate a full trajectory although the ATE is increased significantly and the generated trajectory in the *Co-run* case is far from the ground-truth or that of the *Solo* case. This is mainly because the DNN task is running on a CPU core on the Pi 4 instead of running on the GPU. On the Pi 4, the DNN task running on a CPU core does not generate as much DRAM bandwidth contention

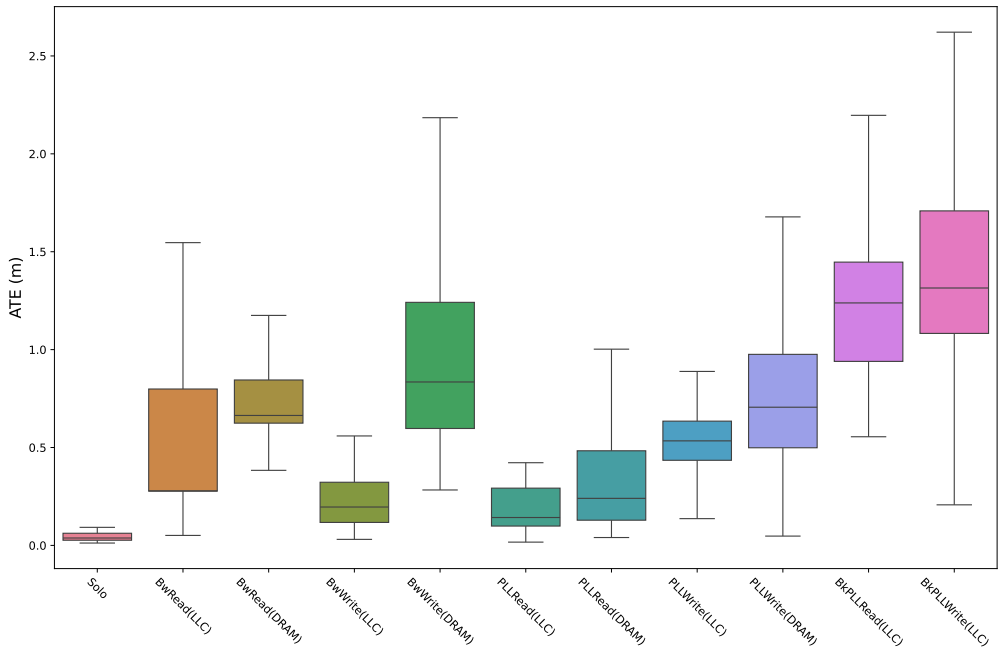
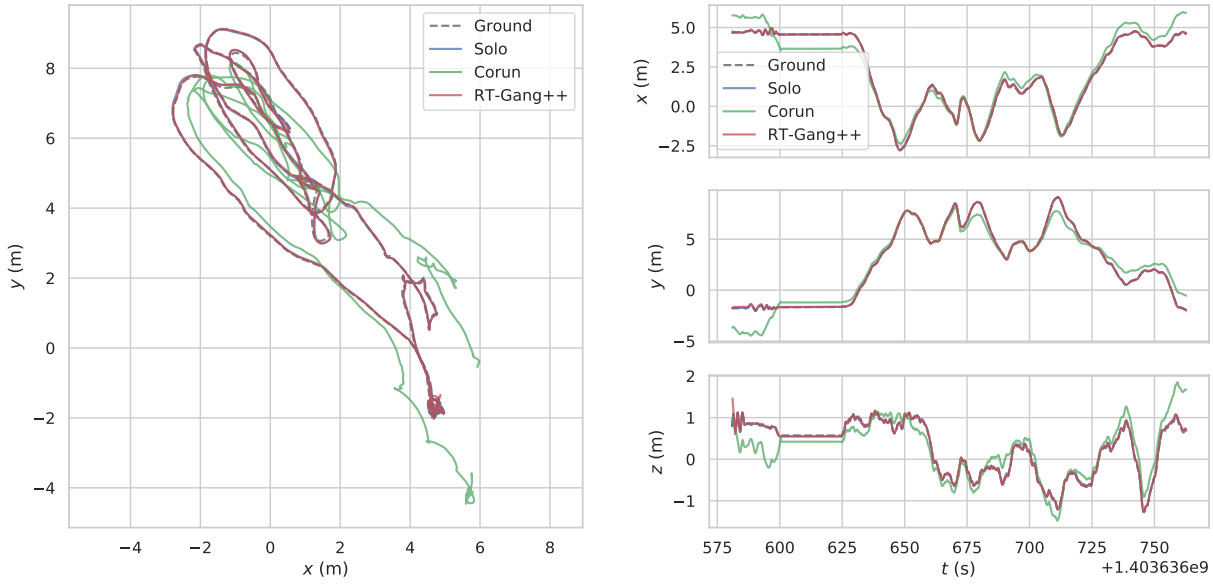


Fig. 7: Impact of DoS attacks on the Absolute Trajectory Error (ATE) of the OV^{2}SLAM generated trajectory on the Pi 4.



(a) Trajectory in X-Y plane

(b) X, Y, and Z positions over time.

Fig. 8: Impact of RT-Gang++ on OV^{2}SLAM performance on the Pi 4.

compared to if it was run on a GPU. Second, RT-Gang++ is able to effectively mitigate the contention generated by the DNN task and DoS attacks and protect the performance of the SLAM task. Concretely, RT-Gang++ reduces the median ATE down to only ∼0.05 meter, which is very close to the ∼0.04 median ATE seen in the *Solo* case.

8 Related Work

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. Notable SLAM algorithms include the LSD-SLAM algorithm [12], the many iterations of the ORB-SLAM algorithm [28], [29], [11], and the OV^{2}SLAM algorithm [15], 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. Recently, Li et. al, observed that performance of SLAM algorithms can be sensitive to execution timing delays and proposed an adaptive strategy within the SLAM to minimize performance degradation [23]. In contrast, our

work proposes a system-level solution that does not require changes in the SLAM algorithm and provides in-depth microarchitectural analysis on resource contention.

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 [31] in shared DRAM controllers [27]. 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 [20]. Woo et al. investigated DoS attacks on bus bandwidth and shared cache space in a simulated environment [39]. Valsan et al. and Bechtel et al. showed that DoS attacks could target internal LLC hardware structures [9], [8], [38]. Based on [9], Iorga et al. presented a statistical approach for testing DoS attacks [19]. Li et al. applied machine learning to better optimize DoS attackers, resulting in WCET slowdowns >400X [24]. 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 [41]. 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 in the real-time systems research community. Partitioning of shared resources, especially shared cache [26], [21], [22], [14], [40], [32] and DRAM banks [22], [14], has been extensively studied. Bandwidth throttling [43], [40], [37], [34], [35], [45], [13] has been another popular approach. MemGuard [43] uses per-core hardware performance counters to throttle each core's bandwidth usage, which has been a standard throttling technique in many subsequent studies. These mechanisms are used to enable tighter worst-case timing analysis on multicore. An exhaustive review on multicore timing analysis can be found in [25]. Recently, both Intel and ARM also introduced hardware support for shared resource partitioning and throttling [18], [5], though their effectiveness in providing isolation for real-time systems is still insufficient [44], [36], [7]. For example, it was reported that even at the maximum throttling level, Intel RDT was not able to protect critical real-time task because the throttled cores were still able to generate significant traffic, enough to cause a 80% performance degradation of the real-time task in the worst-case [36]. Nevertheless, these hardware capabilities orthogonal and can easily be integrated into our framework.

In most of these works, cores are partitioned between RT and best-effort cores. For example, Saeed et al. proposed a memory utilization based dynamic throttling system that protects a single RT core by throttling memory bandwidth usage of the other best-effort cores [34], [35]. Seals et al., also proposed a dynamic throttling system, which also can throttle iGPU's memory bandwidth, to protect a single RT core [13]. However, such an approach can significantly under-utilize computing resources. For example, in the AR-HUD industrial challenge problem, one RT core is not sufficient as the application consists of a multi-threaded CPU task and a GPU task, both of which need real-time guarantees. RT-Gang [2], [1] offers more flexible scheduling potential as all cores can be utilized for both real-time and best-effort tasks. This is because the OS automatically throttles the best-effort tasks only when a RT task is running on any core(s) in the multicore system. However, it does not offer any protection between the real-time tasks on different cores and the iGPU. In this work, we leverage RT-Gang but address its limitations by adding iGPU throttling, CPU cache bandwidth throttling, and partitioned gang scheduling, which allowed us to successfully consolidate the ARM industrial challenge application while ensuring its performance in the presence of fully loaded aggressors on a real heterogeneous multicore platform.

9 Conclusion

In this paper, we presented a solution to the Industrial Challenge problem put forth by ARM in 2022 [3]. 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. To address this we propose RT-Gang++, which combines LLC and iGPU bandwidth throttling to mitigate shared resource contention from their respective resources. By deploying RT-Gang++, we were able to effectively protect the performance of the critical SLAM task, such that it could achieve near solo case performance, without having to over-provision the system.

For future work, we 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 and the generality of the proposed RT-Gang++ framework.

Acknowledgements

This research is supported in part by NSF CNS-1815959, CPS-2038923 and NSA Science of Security initiative contract no. H98230-18-D-0009.

References

[1] W. Ali, R. Pellizzoni, and H. Yun. Virtual gang scheduling of parallel real-time tasks. In *DATE*, pages 270–275. IEEE, 2021.
[2] W. Ali and H. Yun. RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems. In *RTAS*, 2019.
[3] M. Andreozzi, G. Gabrielli, B. Venu, and G. Travaglini. Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm. In *ECRTS*, 2022.
[4] SOAFEE. https://www.soafee.io/.
[5] ARM. *Arm Architecture Reference Manual Supplement: Memory System Resource Partitioning and Monitoring (MPAM), DDI:0598B.b*, 2020.
[6] M. Bechtel and H. Yun. Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems. *Transactions on Computers*, 2021.
[7] M. Bechtel and H. Yun. Cache Bank-Aware Denial-of-Service Attacks on Multicore ARM Processors. In *RTAS*, 2023.

[8] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun. DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car. In *RTCSA*, 2018.

[9] M. G. Bechtel and H. Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *RTAS*, 2019.

[10] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart. The EuRoC Micro Aerial Vehicle Datasets. *The International Journal of Robotics Research*, 2016.

[11] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 2021.

[12] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *ECCV*, 2014.

[13] H. Y. Eric Seals, Michael Bechtel. Bandwatch: A system-wide memory bandwidth regulation system for heterogeneous multi-core. In *RTCSA*, 2023.

[14] F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun. Deterministic Memory Abstraction and Supporting Multicore System Architecture. In *ECRTS*, 2018.

[15] M. Ferrera, A. Eudes, J. Moras, M. Sanfourche, and G. Le Besnerais. OV^{2}SLAM: A Fully Online and Versatile Visual SLAM for Real-Time Applications. *IEEE robotics and automation letters*, 2021.

[16] HopeNet-Lite. https://github.com/OverEuro/deep-head-pose-lite.

[17] rosbag2. https://github.com/ros2/rosbag2.

[18] Intel. Intel® Resource Director Technology (Intel® RDT) Framework. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[19] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson. Slow and Steady: Measuring and Tuning Multicore Interference. In *RTAS*, 2020.

[20] G. Keramidas, P. Petoumenos, S. Kaxiras, A. Antonopoulos, and D. Serpanos. Preventing denial-of-service attacks in shared cmp caches. In *SAMOS*, 2006.

[21] 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.

[22] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith. Attacking the One-Out-of-M Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. *Real-Time Systems*, 2017.

[23] A. Li, H. Liu, J. Wang, and N. Zhang. From Timing Variations to Performance Degradation: Understanding and Mitigating the Impact of Software Execution Timing in SLAM. In *IROS*, 2022.

[24] A. Li, M. Sudvarg, H. Liu, Z. Yu, C. Gill, and N. Zhang. PolyRhythm: Adaptive Tuning of a Multi-Channel Attack Template for Timing Interference. In *RTSS*, 2022.

[25] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[26] 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.

[27] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security Symposium*, 2007.

[28] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. ORB-SLAM: A Versatile and Accurate Monocular SLAM System. *IEEE transactions on robotics*, 2015.

[29] R. Mur-Artal and J. D. Tardós. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE transactions on robotics*, 2017.

[30] NVIDIA. *Tegra X1 Mobile Processor technical reference manual (revision 1.3p)*, 2019.

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

[32] S. Roozkhosh and R. Mancuso. The Potential of Programmable Logic in the Middle: Cache Bleaching. In *RTAS*, 2020.

[33] N. Ruiz, E. Chong, and J. M. Rehg. Fine-Grained Head Pose Estimation Without Keypoints. In *CVPR*, 2018.

[34] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In *RTAS*, 2022.

[35] A. Saeed, D. Hoornaert, D. Dasari, D. Ziegenbein, D. Mueller-Gritschneder, U. Schlichtmann, A. Gerstlauer, and R. Mancuso. Memory latency distribution-driven regulation for temporal isolation in mpsocs. In *Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[36] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger. A Closer Look at Intel Resource Director Technology (RDT). In *RTNS*, pages 127–139, 2022.

[37] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: a System-wide Framework for Memory Bandwidth Profiling and Management. In *RTSS*, 2020.

[38] P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *RTAS*, 2016.

[39] D. H. Woo and H. Lee. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In *CMP-MSI*, 2007.

[40] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic Resource Allocation for Multicore Real-Time Systems. In *RTAS*, 2019.

[41] T. Yandrofski, J. Chen, N. Otterness, J. H. Anderson, and F. Smith. Making Powerful Enemies on NVIDIA GPUs. In *RTSS*, 2022.

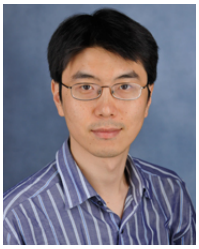
[42] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *RTAS*, 2014.

[43] 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.

[44] M. Zini, D. Casini, and A. Biondi. Analyzing arm's mpam from the perspective of time predictability. *IEEE Transactions on Computers*, 72(1):168–182, 2022.

[45] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso. MemPol: Policing Core Memory Bandwidth from Outside of the Cores. In *RTAS*, 2023.

Michael Bechtel received a B.S. degree in Computer Science from the University of Kansas in 2017 and a Ph.D. degree in Computer Science from the University of Kansas in 2023. His research interests include real-time embedded systems, computer architecture, and cyber-physical systems. His work has appeared in top embedded real-time systems venues such as RTAS, and he received an Outstanding Paper Award from RTAS'19. He is currently a software engineer at Garmin.



Heechul Yun is an associate professor in the department of Electrical Engineering and Computer Science at the University of Kansas. His research interests include OS, computer architecture, and real-time embedded systems with special emphasis on addressing real-time, security, and safety related issues on safety-critical cyber-physical systems (e.g., autonomous cars and UAVs). His work has appeared in top embedded real-time systems venues such as RTAS, ECRTS and Transactions on Computers; and received multiple prestigious paper awards (Best Paper Award from RTSS'20, Outstanding Paper Award from RTAS'19, Best Paper Award from RTAS'16, Editor's Pick of the Year Award from IEEE Transactions on Computers in 2016). He received a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 2013. Prior to his Ph.D., he worked at Samsung Electronics as a senior software engineer.