RT-ADAS Report

Devon Lewis, Melody Chan, Kolade Adegbaye

Supervised by Renato Mancuso

Abstract—The goal of the directed study at first was to work on ECRTS22's Industrial Challenge 2022 [1] . The challenge involved analyzing high-compute real-time autonomous vehicle workloads on an ARM SoC system, specifically an NVIDIA Jetson Xavier chip. Since the analysis for the case study was open-ended, we decided to apply multi-core inference strategies developed in part by our Professor, to the Jetson. A large portion of his research involves improving predictability in real-time multicore embedded systems.

This report describes and summarizes the work that was completed during the directed study, as well as describes at a high level the general research space we worked in.

1 Introduction

Real-time systems have an important temporal element to them - they must complete tasks by a certain deadline. We can further categorize them based on the importance of missing a deadline. Safety-critical systems are such that missing a deadline is catastrophic and must be avoided at all costs, for example, airbag controllers. An embedded chip with this software must be certified, a guarantee that no deadlines will be missed for the lifetime of the chip. In order to assert the guarantee, system designers focus on predictability. Predictability is the difference between the worst-case execution time and best case execution time of a task. In general, the goal is to minimize the difference to obtain a tight bound on execution time.

Predicting execution time in general is a very difficult problem due to the complexity of modern hardware. Because of this, almost all safety critical real-time systems are single core, or multicore with all but one core activated. This is unfortunate, since the performance benefits of multi-core systems are then lost. In order to address this problem, much research has been done to improve the predictability of these multicore systems in hopes of using them for real-time tasks and getting them certified for safety critical workloads.

One of the biggest contenders for unpredictability in multicore systems is the memory hierarchy, in particular shared caches. The reason for this is that two or more tasks running on separate cores can modify the same cache, thereby evicting data needed by the other tasks. We focused on two techniques to reduce shared cache contention: cache partitioning and cache locking. Our Professor's work involves an implementation of cache partitioning based on the Jailhouse hypervisor that we leveraged during this directed study.

In sum, our goal was to deploy this modified Jailhouse on the Jetson and measure its effectiveness at mitigating contention. Unfortunately, the actual chip took some time to get here, so we had to change course and use an older chip, the NVIDIA Jetson TX1.

To verify the effectiveness of the contention mitigation techniques, we were to familiarize ourselves with the benchmark suite RT-Bench [2]for deployment on the TX1 and eventually the Xavier, as a precursor to deploying the benchmarks specified in the challenge. By tackling these tasks, we gained experience developing on and using embedded systems. This experience included but was not limited to communicating over a serial connection, flashing firmware, cross-compiling a Linux kernel, modifying boot sequences, and investigating data sheets and manuals.

After running experiments with the benchmark, we needed to compile and load Jailhouse for the TX1. Here we encountered numerous problems which are described in the following sections.

To supplement the practical work, we read a few papers [1] relating to predictability in embedded systems as well as cache mitigation techniques. Most meetings were supplemented by a mini-lecture describing either the papers or related general concepts like virtualization and hypervisors.

2 Research Background

The following section will describe mitigation strategies discussed above. Basic knowledge of cache geometry and computer systems is assumed.

2.1 Cache Locking

Cache locking prevents specified cache lines from being evicted by different cores [3]. Unlike cache partitioning, there must be hardware support. Cache locking reduces inter-core interference, that is, the contention that results from tasks of different cores evicting each other from a shared cache.

2.2 Cache Partitioning

Cache partitioning assigns a certain portions of a cache to a task or a core [3]. Techniques are divided into index and way-based partitioning, and each is further subdivided into hardware and software based approaches. Hardware support does not exist for all embedded chips, including the Jetson TX1 and Xavier. Software based approaches hold a clear advantage with respect to availability, and in general require careful OS modifications to the virtual memory subsystem. Briefly, index-based partitioning methods allocate associative sets, while way-based partitioning allocates entire ways. In general, software approaches to way-based partitioning are rare and way-based methods are directly coupled with the associativity of the cache. Increasing the associativity of the cache to increase the performance of way-based methods is typically infeasible.

2.2.1 Page Coloring

Page coloring is an effective index-based software approach to cache partitioning. In modern virtual memory systems, the virtual address of data in memory contains the information needed for a cache lookup. In particular, it contains the set index. Controlling the set index bits would allow control over which set, and subsequently which way, data can reside. That control is precisely the goal of page coloring. By restricting a task or core from accessing all but its assigned sets, we remove its ability to evict cache lines from other sets.

The fundamental unit of data in virtual addressing schemes is a page, typically 4KiB. For most caches, multiple pages can fit into one cache way. It is also the case that a given way spans multiple sets. Therefore, we can assign each page a color such two pages with different colors always map to different sets.

2.2.2 Virtualization-based Coloring

The implementation of page coloring requires non-trivial modifications to the OS, and as such is not always desirable. A solution explored by our Professor leverages virtualization by a hypervisor (Jailhouse) to control the virtual address manipulation [4]. Using a hypervisor however, comes with the issue of portability, as configurations may change depending on the embedded chip it runs on. In fact, we had to apply small edits to Jailhouse in order to get it working on the TX1. We expect more edits when we attempt to use Jailhouse on the Xavier.

1. Papers not explicitly cited will be in the bibliography.

3 RT-Bench

As described above, RT-Bench is a real time extension to popular compute intensive benchmarks. The ones we specifically used were its image computing benchmarks. They process both disperse and full-HD images that differ in workload intensity through some common image computing algorithms. The workloads have been modified to run periodically. It also has logic to produce CSV outputs and record information such as deadline misses and response time.

3.1 Experiments

We were interested in measuring the effect of inter-core interference on the predictability of various multi-core processors with respect to the benchmark workloads.

3.1.1 *Local Machine Outputs*

Before we were able to configure and use the Jetson TX1, we ran the experiments on our local machine, which has a 4-core Intel Core i5 processor. On the local machine, we were able to run the benchmarks on both disperse and full-HD images without deadline misses. In this report, we included the output graphs 1 and 2 from running the rt-benchmarks on disparity images, different in the amount of memory allocated to tasks. The blue indicates the baseline test without contention, while the orange one indicates the test with *bandwidth* enabled.

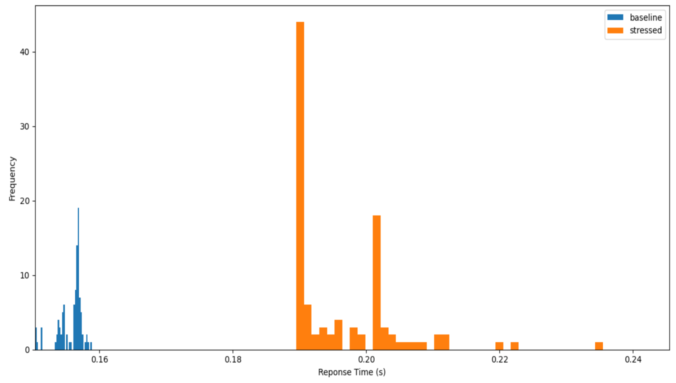

Fig. 1: *disparity* baseline and m25600 bandwidth contention on local machine

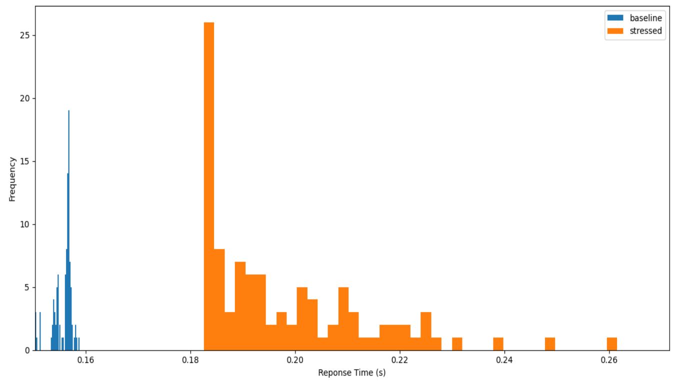

Fig. 2: *disparity* baseline and m51200 bandwidth contention on local machine

From the output graphs we can see the amount of memory allocated for each task effects both response time and predictability of the running tasks. The more memory that is allocated to the tasks, the wider it spreads on the x-axis, and therefore the more unpredictable it is.

3.1.2 *Jetson TX1 Outputs*

Going over the same steps as we did on the local machine, the Jetson TX1 appeared to be overloaded by the full-HD images as it missed most of the deadlines. We decided to run only the disperse images on TX1 because with deadline misses, we won't be able to obtain much useful information. We had to adjust some parameters relating to the size of the memory accesses in order to target a shared cache of the TX1 as opposed to a private L1 cache. Figure 3 depicts the response time of the disparity benchmark running 100 times on Jetson TX1 with both m25600 and m51200 allocated memory. Again, blue indicates the baseline without any contention, orange and green are the stressed results.

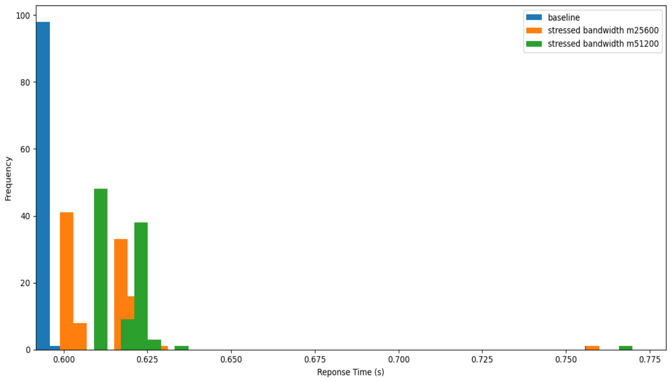

Fig. 3: *disparity* baseline and bandwidth contention

Two conclusions may be drawn from the figures. Firstly, the interference increases response time by about half a second in this instance. Secondly, predictability decreases when there is interference, indicated by a larger range of values. We can clearly see a need for mitigation strategies.

4 Jailhouse

Most of our time was spent working on getting the Jailhouse fork to run on the TX1. We were indeed able to get it working, but did not have time to activate the cache partitioning mode and record results. Despite that, we learned quite a bit.

4.1 Background

Jailhouse is a partitioning hypervisor, meaning that it focuses on isolation rather than virtualization. As such, it is lightweight, but lacks common hypervisor features like device emulation. It splits the system up into isolated units called cells, and each cell has some resources allocated to them such as CPUs or memory. Tasks that require full control over a CPU for example, could benefit from this isolation. We will see an example of such a task shortly. Also, Jailhouse operates in a physically continuous memory region that must be reserved at boot using the *mem* keyword in the extlinux.conf kernel entry.

4.2 Building the Kernel

The jailhouse fork is compatible with Linux 4.9+ but the version running on the TX1 was older. As such, we needed to build the *linux-tegra-4.9* [2] kernel from source and enable it on the TX1. We decided to cross-compile it on our workstation targeting the aarch64 architecture and then send the kernel image over

2. The NVIDIA downstream Linux kernel source for the Jetson platforms

scp. For our first attempt, we simply renamed the old kernel image so it would not be selected by the bootloader (U-Boot). Upon restart, we were unable to ssh into the TX1 nor could we see anything over our USB serial connection. We had to restart our process by reflashing the TX1 using NVIDIA's JetPack SDK.

Our second attempt was more principled. Rather than renaming the old kernel, effectively destroying any backup, we created a new entry in extlinux.conf. With this, we could now choose to boot from either kernel with U-Boot. After rebooting, we were able to get access to the TX1, but we had no opportunity to switch the kernel. By default the TX1 was booting into the wrong kernel.

Until that point, the serial connection over USB was sufficient, and we were able to get shell access post-boot. However, we needed access pre-boot, to be able to interface with U-boot and choose the correct kernel. We required a more traditional serial connection, and used a UART to TTL cable. After combing through the manual to find which pins allowed a serial connection, we tried another reboot. Finally we were able access U-Boot and launch the correct kernel.

4.2.1 Building Jailhouse

Jailhouse operates as a kernel module, so we had to build its kernel object file. That process was straightforward. Before running it however, we needed to configure it to work with the TX1.

4.3 Configuration

Jailhouse must be configured for the particular hardware it will run on. The configuration files define the memory locations Jailhouse may *not* map to, among other things. It was already ported to the Jetson TX1, so we foresaw no difficulties. However, we got a memory access violation error when starting it up. We immediately thought it was related to the amount of contiguous memory we carved out for Jailhouse - perhaps it was not enough. After provisioning more, the error persisted.

We then began combing through the configuration files, looking for any defined memory regions. They were located in the *jetson-tx1.c* configuration files. Since the error gave us the specific address accessed, we were able to zero in on the deallocated region. Strangely enough, there was a gap in the memory regions. Jailhouse was accessing a region it should not have had access to, so the kernel shut it down. The solution was simple; we added an entry in the range of addresses of sufficient size. We tried once more to activate Jailhouse, and we received a different error. Progress.

4.4 MemGuard

The error was related to MemGuard, a memory bandwidth reservation system [5]. Along with cache partitioning, the fork also supports MemGuard. Combined, both approaches allow for fine-grained control of the memory subsystem. For reasons we could not fully understand, MemGuard was incorrectly initializing the interrupt controller, preventing Jailhouse from starting. We simply disabled it, and Jailhouse successfully launched.

4.5 Memory Bombs

To get acclimated to Jailhouse, we were instructed to deploy a demo task that was already present in the repository. Similar to the bandwidth benchmark in RT-bench, the memory bomb task repeatedly reads and writes to memory. The difference here is that Jailhouse will dedicate an entire CPU to running this application. The goal was to once again measure contention, but this time with the memory bombs running rather than the benchmark. This setup is closer to how actual workloads would operate on a multicore system. Unfortunately, at this point the semester had concluded and we were not able to take the measurements

5 Conclusion

The directed study was a great introduction to working with embedded systems. It furthered our Linux knowledge as well, with tasks such as building kernels and working with external devices. Although we were not able to complete our original goal of deploying benchmarks on the Xavier, we can certainly continue our work in the future.

Appendix A
RT-bench

```
./disparity -p 1 -d 0.3 -t 100 -c 3 -f 99 \
-m 10M -o out.csv -l 2 -b .
```

Listing 1: Running disparity to generate a CSV

```
import pandas as pd
import matplotlib as plt

data_norm = pd.read_csv(
    r'outputs_disparity/output.csv')

data_stress = pd.read_csv(
    r'outputs_disparity/stress.csv')

response_time_norm =
data_norm['job_elapsed(seconds)']

response_time_stress =
data_stress['job_elapsed(seconds)']

ax = response_time_norm.hist(
    cumulative=False, bins=40)

ax.set_ylabel("Cumulative Frequency")
ax.set_xlabel("Reponse Time (s)")
```

Listing 2: Snippet of preprocessing and visualization code

Appendix B
Jailhouse

```
insmod driver/jailhouse.ko
jailhouse enable \
./configs/arm64/jetson-tx1.cell
```

Listing 3: Adding Jailhouse as a kernel module and starting it

```
jailhouse cell create \
./configs/arm64/jetson-tx1-bomb.cell
jailhouse cell load 1 \
./inmates/demos/arm64/mem-bomb.bin
jailhouse cell start 1
```

Listing 4: Loading and executing the memory bomb

References

[1] M. Andreozzi, G. Gabrielli, B. Venu, and G. Travaglini, "Industrial Challenge 2022: A High-Performance Real-Time Case Study on Arm," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, M. Maggio, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 231, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 1:1–1:15, ISBN: 978-3-95977-239-6. DOI: 10.4230/LIPIcs.ECRTS.2022.1. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16318.

[2] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications," *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022. DOI: 10.1145/3534879.3534888.

[3] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys*, vol. 48, no. 2, pp. 1–36, 2015. DOI: 10.1145/2830555.

[4] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[5] L. Chen and Z. Zhang, "Memguard: A low cost and energy efficient design to support and enhance memory system reliability," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 49–60. DOI: 10.1109/ISCA.2014.6853221.

[6] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, S. Quinton, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 133, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 27:1–27:25, ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.27. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10764.

[7] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters, "Identifying the sources of unpredictability in cots-based multicore systems," *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013. DOI: 10.1109/sies.2013.6601469.