Performance analysis of a virtualized vehicle-compute platform: An experience report

Christopher Hesse, Tim Welsch
{christopher.hesse, tim.welsch}@aptiv.com
Aptiv, Hildesheim, Germany

Holger Eichelberger
eichelberger@sse.uni-hildesheim.de
University of Hildesheim, Germany

Abstract

Compute platforms for modern automotive systems tend to combine embedded properties, increasingly complex architectures and even virtualization. However, analyzing the performance of such systems, e.g., to identify performance bottlenecks, is not trivial.

In this paper, we report our experience in analyzing the performance of a camera-vision application on a virtualized vehicle-compute platform. We discuss issues that we faced during the analysis, impacts of the virtualization on the performance as well as causes.

1 Introduction

In upcoming vehicle architectures, a plethora of parallel running, but physically separated electronic control units tend to be replaced by centralized, complex compute platforms [2]. For various reasons, e.g., to separate safety domains [2], virtualization is desirable on such platforms. Although modern processors provide virtualization support, running and separating multiple virtual domains on the same processor causes overhead. This overhead may impact the applications, in particular if (soft-)realtime processing is required.

Aptiv developed the Connected Server Platform (CSP) as a technology demonstrator for the next generation architecture on head units for cockpit computing. The aim is to serve all cockpit and cabin functionality on a single platform while separating functionality in (virtualized) domains. One specific use case of CSP is to run computer vision algorithms such as face recognition, eye gaze detection, background segmentation within multiple cockpit, cabin monitoring and infotainment services. As a requirement, CSP shall render 30 frames per second (fps). However, the virtualized CSP lead to unexpected performance issues, e.g., a stuttering or flickering display as well as a general impression of a slower system. No issues were noticed when running CSP without virtualization.

While it is not surprising that virtualization may cause a noticeable overhead, a more detailed analysis is needed to detect the root causes. In this paper, we report on a measurement-based performance analysis of the CSP. Based on a cause tree of potential reasons, we define a set of metrics that can be attributed to each reason and apply systematic experimentation to analyze the causes. We identify the overhead of virtualized CPU- and GPU-based rendering, issues in the virtualization setup as well as problematic system services. Related work usually focuses on comparison of virtualization approaches [1] or on GPU virtualization [3] rather than their combination (in an embedded system). We believe that our results can help developing and improving similar systems.

Structure of this paper: In Section 2, we introduce our approach and the metrics. We detail our setup for the experiments in Section 3 and discuss the obtained results in Section 4. Finally, in Section 5, we conclude and provide an outlook on future work.

2 Approach

In this section, we describe our approach[1] for a performance analysis of the CSP. We start with a system description, discuss then potential causes for performance issues and runtime metrics to trace the issues.

On the hardware side, CSP is based on standard consumer components, in particular three AsRock Z270 mainboards, each equipped with an Intel® Core i5-7600 (Kaby Lake) processor, integrated GPU 630, 16 GB memory, and a Samsung SSD SM961-NVMe 128 GBytes. CSP utilizes multiple cameras as input, which are connected through USB ports. On the software side, we use as operating system Yocto Linux[2], a popular Linux variant for embedded systems. For virtualization, we use Xen[3], a hardware-based (type 1) hypervisor with XenGT[4] supporting for the Intel® GPU. In the virtualized setup, a privileged domain hosts the hardware drivers, while guest domains run the applications and indirectly access the hardware such as the GPU through the privileged domain.

Inspired by [4], we build a cause tree for potential performance issues based on the involved components. Figure 1 depicts a simplified version of the cause tree.

The CSP can be set up directly on the hardware/ operating system (bare metal) or may be subject to **virtualization**. In the virtualized setup, the number of domains and their assignment to (virtual) CPU

[1] More details, e.g., the underlying BSc thesis will be made available before publication.

[2] https:/ www.yoctoproject.org

[3] https://www.xenproject.org

[4] https://github.com/intel/XenGT-Preview-xen

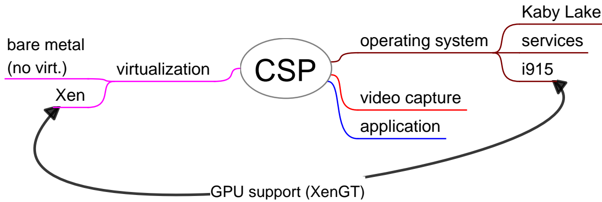

Figure 1: Cause tree for performance issues in CSP.

cores may affect the performance of the CSP. Here, the idea is to compare results for the same metrics for bare metal and (different) virtualized setups.

The fundamental parts of CSP are the operating system, the video capture through the cameras as well as the application executing the computer vision algorithms. The Yocto **operation system** is mostly used without modifications. Specific drivers for the *Kaby Lake* processor as well as its GPU (*i915*) are required as otherwise the CSP cannot boot. Operating system services and their configuration may be a reason for performance problems [4]. The **video capture** links the cameras and the CSP application by a stream of video frames. In the virtualized setup, passing the video stream from the USB driver through the privileged domain to the guest domains may cause performance issues. Also the configuration of the CSP **application** may impact the performance, e.g., if CPU rendering is used instead of the GPU. In virtualization, GPU rendering is done by the privileged domain, i.e., inefficient communication among the guest and the privileged domain may affect the performance. Due to space limitations, the SSD is out of scope here.

To measure the performance impact for the potential causes, we apply the metrics summarized in Table 1 including system (load and memory use), video, GPU and graphic benchmark-specific metrics.

3 Experimental Setup

Now, we detail the experiment setup, i.e., we discuss the installation(s) of CSP for measurement, the measurement tools, the measurement procedure and the analysis of the gathered data.

We use the original hardware of the CSP system as described above. However, we utilize Onsemi AR0144 monochrome cameras, because the original cameras are highly sensitive to ambient lighting, i.e., the input frame rate was massively fluctuating. This is acceptable for our setup, as the replacement cameras lead to representative and stable video streams.

Cause	Metric type
operating system	system & process load, system & process memory use
video capture	fps, system & process load
application	fps, system & process load
GPU	busy, active, stall, gl2mark score

Table 1: Metric types for detecting potential causes.

Regarding the software, we set up several installations of the CSP, including variants for bare metal and virtualization, all based on Yocto Linux 2.3 Pyro, Xen 4.6 with one guest domain. Different versions of Xen or Yocto may lead to different results. However, due to technical dependencies, e.g., to the hardware, only some combinations can be executed at all on the CSP and, thus limit the subjects in the experiment. To consider the impact of the installed software, we prepare a fully patched native variant (FPN) with all Aptive patches on bare metal, a partially patched variant only with Kaby Lake and i915 support (PPN) as well as similar virtualized variants for Xen (FPX, PPX).

Most of the metrics can be obtained from the operating system via `/proc`. For measuring the frame rate, we developed a simple benchmarking application. For obtaining the GPU metrics, we use GPUTop[5] as alternatives such as Intel® VTune[6] or intel_gpu_top[7] are either not available or only scarcely documented. However, in a virtualized setup, the GPU performance counters are not accessible in a guest domain as they are not passed through by the i915 driver and XenGT. Thus, we obtain the GPU measures only for the privileged domain and mirror them into the guests. To analyze the GPU rendering, we use *glmark2*[8], an OpenGL graphics benchmark for embedded systems.

For measuring the performance of the CSP application, instrumentation of the original C code would be required, e.g., to track the number of rendered frames. Moreover, the application performs asynchronous rendering and includes functionality that is not required for the experiments. To avoid unintended disturbances, but also to experiment with on/off screen CPU/GPU frame processing and rendering, we use a representative simplified single-threaded benchmark application called *opencv_mog2*, which is based on the code of the original CSP application, performs similar computations and collects the required metrics.

We obtain measurements by querying metrics/running the benchmarks as a script on the system variants. We execute the script for one hour and collect/store the measurements in a file once per minute as some of the (system) metrics are not updated more frequently. Before running the script, we reboot the CSP and between two subsequent benchmarks, we apply a waiting time of 1 minute for cool down.

For data analysis, we apply Python scripts to preprocess the data and to calculate descriptive statistics. For visualizing the results, we use gnuplot, e.g., to draw boxplots, histograms or time series diagrams.

[5]https://github.com/rib/gputop
[6]https://software.intel.com/en-us/intel-vtune-amplifier-xe
[7]http://cgit.freedesktop.org/xorg/app/intel-gpu-tools
[8]https://github.com/glmark2/glmark2

4 Experimental Results

In this section, we summarize our results.

For the **operating system**, we observed the CSP application in CPU-only rendering mode. As indicated in Figure 2, the performance loss in terms of rendered fps between bare metal (FPN) and the default FPX setup at Aptive with 2 virtual CPU cores for each, privileged and guest domain, is about 36%. Allocating 4 virtual CPU cores to the guest domain (irrespective whether the privileged domain receives 2 or 4 virtual cores) reduces the performance drop to 24%. Despite the performance differences, process and system load are roughly the same for FPN/FPX with 4 virtual cores for the privileged domain. In contrast, the FPX configurations with 2 virtual cores for the privileged domain increases both, process and system load by 4% up to 21%. This is caused by continuously rendering guest requests in the privileged domain and leads to a bottleneck at low resources. However, we did not notice significant differences between FPN/PPN or FPX/PPX, respectively.

While monitoring the memory consumption of all processes over time, we observed a slightly increasing memory use of the operating system. As cause, we identified a mis-configured `systemd` service and potential memory leaks in a `zabix` monitoring daemon.

The performance difference for the **video capture** was not significant, i.e., the variants achieved a mean of 25 frames/s at a standard deviation of 0.3.

Due to these results, we focus for the remaining discussion on the FPN and the FPX configuration with 4 virtual cores for privileged and guest domain. All configurations for on/off screen CPU/GPU frame processing and rendering of the **application** caused a performance loss between FPN and FPX of around 25%, e.g., the CPU-only application renders in average around 88 fps on FPN and 66 fps on FPX. Typically, system and process load are rather similar for the FPN and FPX setup. However, a more detailed analysis shows that the FPX setup produces more load variations and even spikes that we can attribute to stolen and I/O waiting caused by the rendering communication between the domains.

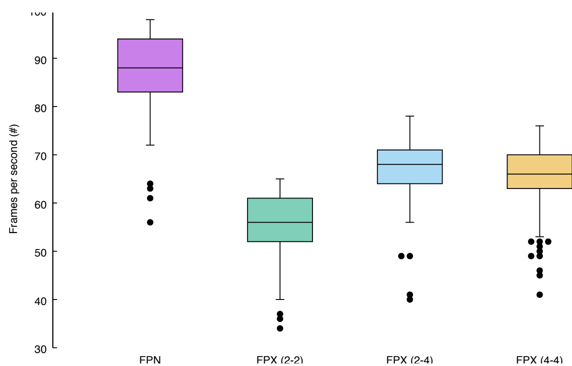

Figure 2: Frames/s in opencv_mog2.

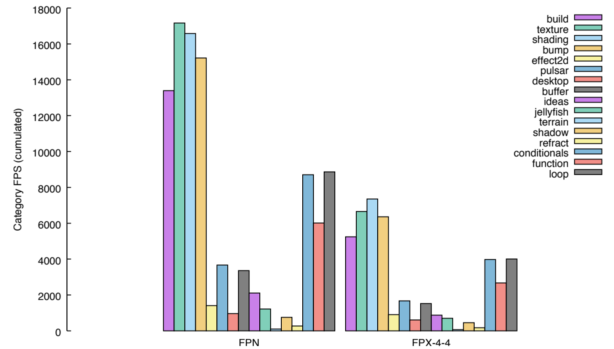

Figure 3: glmark2 off-screen results.

Regarding **GPU** capabilities, we compare the results of glmark2 for FPN and FPX. We run glmark2 for both, on-screen and off-screen rendering. Figure 3 illustrates the off-screen rendering for FPN and FPX. Regarding the overall glmark2 score, the performance loss by virtualization is here 57% for off-screen and 62% for on-screen rendering, i.e., more demanding rendering also leads to a higher performance impact.

5 Conclusions and Future Work

Developing embedded, virtualized automotive architectures is challenging and systematic performance measurements can significantly support system development. In this paper, we analyzed the performance of the Aptiv CSP. The measurements indicate that virtualization allows running a demanding application with the required performance, while high demanding loads can lead to a performance loss of more than 50%. Moreover, we identified a better setup of the virtualization and of some system services. As we utilized available components and system measures, we believe that our approach can be applied to similar systems, even outside the vehicle domain.

In the future, we aim at experiments with more recent versions of Yocto and Xen. We also envision to integrate measurements into the Aptiv development, e.g., to detect performance regressions early.

References

[1] J. Hwang et al. “A component-based performance comparison of four hypervisors”. In: *IM’13*. May 2013, pp. 269–276.

[2] C. Patsakis, K. Dellios, and M. Bouroche. “Towards a Distributed Secure In-vehicle Communication Architecture for Modern Vehicles”. In: *Comput. Secur.* 40 (2014), pp. 60–74.

[3] H. Chen et al. “GaaS workload characterization under NUMA architecture for virtualized GPU”. In: *IISPASS’17*. 2017, pp. 65–76.

[4] H. Knoche and H. Eichelberger. “Using the Raspberry Pi and Docker for Replicable Performance Experiments: Experience Paper”. In: *ICPE’18*. 2018, pp. 305–316.