Leveraging Data Center Architectures for Full Flight Simulators

Jean-Philippe Arbic, Ghislain Boivin, Nick Giannias
CAE inc.
Montreal, Quebec
jean-philippe.arbic@cae.com, ghislain.boivin@cae.com, nick.giannias@cae.com

ABSTRACT

Aircraft simulation for pilot training in an immersive environment presents some unique challenges. High fidelity aircraft performance and system models are required to replicate normal and abnormal conditions; human pilots-in-the-loop require high computational frequencies to replicate control feel; aircraft communication busses have seen a dramatic increase in the amount of data. These are just some examples that must be met by a real-time hardware and software architecture to prevent noticeable delays, jitter or other artifacts that could negatively impact training.

Traditionally, individual training devices rely on dedicated hardware for their real-time simulation platform. This approach presents significant limitations in terms of flexibility, scalability, and lifecycle costs. Cloud and on-premise data centers have addressed some of these limitations using high density servers, virtualization and containerization. While these technologies are not new to modeling and simulation, this paper discusses how we have adapted and leveraged them successfully on full flight simulators.

Our approach uses commercially available, enterprise-grade hardware and software, typically used for data centers and integrated as the main computational platform. We will share lessons learned in the process, including performance measurements that precisely describe our real-time requirements and how we were able to achieve them by adapting our chosen hardware/software platform. We will also present the results of our experiments in the use of containers without compromising real-time performance. The outcome is our first full flight simulator qualified to ICAO Level 7[1] (ICAO, 2015) with this new architecture which is currently installed and operating at a training center.

ABOUT THE AUTHORS

Jean-Philippe Arbic is Technical Leader in DevOps architecture at CAE, with over 15 years of experience specializing in the architecture, development, and integration of software for real-time simulators. He actively contributes to engineering innovations, particularly in the development of next-generation training devices and the acceleration of product development. Jean-Philippe holds a bachelor's degree in Mechanical Engineering, specializing in Mechatronics, from Ecole Polytechnique de Montreal.

Ghislain Boivin is a Subject Matter Expert - Real-Time Software and Networks at CAE, with over 30 years of software development expertise. His extensive experience for designing low-latency and high-performance software allows him to contribute in the development of critical components for cutting-edge training devices. He now focuses on virtualization technologies, applying his extensive knowledge in this area. Ghislain holds a bachelor's degree in Computer Science from Université de Sherbrooke.

Nick Giannias is a Senior Technical Fellow working for the Enterprise Architecture and Incubation group at CAE and the Chief Architect for CAE's simulator products. He specializes in software development and real-time simulations for concept development, analysis, mission rehearsal, and training applications. Nick holds a bachelor's degree in Mechanical Engineering (Hons.) specializing in Aerospace from McGill University.

[1] Equivalent to FAA/EASA Level D

Leveraging Data Center Architectures for Full Flight Simulators

Jean-Philippe Arbic, Ghislain Boivin, Nick Giannias
CAE inc.
Montreal, Quebec
jean-philippe.arbic@cae.com, ghislain.boivin@cae.com, nick.giannias@cae.com

INTRODUCTION

The use of software for modeling aircraft and their environments, aiming to provide immersive training experiences for future pilots inside full flight simulators, presents unique challenges in the realm of software development. This complexity arises from the intricate balance between the need for high computational power and real-time processing.

At one end of the spectrum, the system is computationally intensive, as the modeling of physical systems necessitates high frequency to maintain the stability of the simulation models. Modern aircraft are also equipped with an extensive array of sensors, and avionic communication protocols such as AFDX[2], necessitating high data throughput with minimal latency. At the other end of the spectrum, the requirement is that these computations must be executed in real-time. Any delay in computation may be perceptible to the crew of a training device or trigger faults in some of the timing sensitive subsystems being simulated.

One of the most critical systems that rely on precise timing and real-time performance is the "flight loop" or human-in-the-loop. The flight loop includes human interaction with the flight controls, calculation of the aircraft response and the resulting motion and visual sensory outputs. In our architecture, this loop includes a hard real-time edge computer that controls the physical motors for control loading and motion and synchronizes with the simulation.

The flight loop is so important to training that civil aviation authorities specify the required performance in order to qualify a simulator for training. This response time is referred to as *transport delay* and it must be minimized to provide realistic response cues and also avoid simulator sickness. It must also be stable, minimizing any variation between computational frames. The ICAO qualification standard for flight simulators (ICAO, 2015) specifies a maximum response time of 120ms from pilot command to a change in the visual scene and 100ms for motion and instrument feedback. Figure 1 illustrates this execution sequence. Part of the transport delay is due to hardware components like the visual projector while others are due to the software models. The part of the delay due to the software models and network data transfers we refer to here as the *simulation delay*.

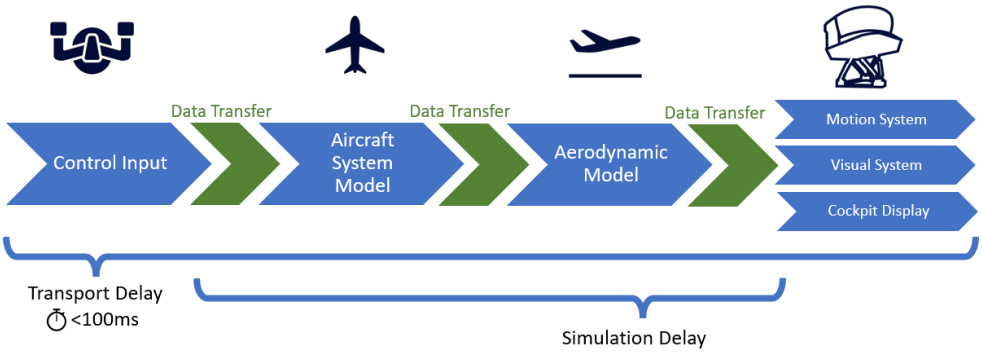

Figure 1. Transport delay and simulation delay in the flight loop execution sequence

[2] Avionics Full-Duplex Switched Ethernet, a specific implementation of ARINC specification 664

The traditional architecture for meeting these real-time requirements involves running simulator software on multiple bare-metal computers over the same network, referred to as the native architecture in figure 3. This solution however introduces its own set of challenges. The setup and maintenance of multiple computers, coupled with the significant space they occupy within a training center, add complexity and significant maintenance cost to simulator operators. Also, this approach not only necessitates the management of hardware obsolescence but also demands considerable effort to adhere to best practices in cybersecurity when dealing with multiple bare-metal instances.

In response to these challenges, the industry is progressively shifting towards a higher degree of abstraction between the software and the hardware on which it operates (Knobbout, 2023) (Moy, 2023) (Kirkemdoll, 2022). This paper will detail how we successfully implemented the simulation of a full flight simulator using virtualization and containerization, and discuss the challenges encountered during the implementation of these technologies.

HARNESSING VIRTUALIZATION AND CONTAINERIZATION

Containerization and virtualization are complementary abstraction technologies that can be used together or standalone to enhance efficiency, scalability, and flexibility. Virtual Machines (VMs) provide strong isolation between applications and the underlying computation hardware. They also provide a significant advantage by simplifying the migration to a different environment, commonly known as "lift and shift". However, this isolation comes with a performance overhead. Containers are another alternative, offering lightweight isolation, and allowing multiple applications to run on the same operating system (OS) instance without the overhead of separate OS instances.

Applying abstraction technology to a flight simulator training device offers multiple potential benefits:

- Higher density, centralized computing hardware helps minimize the overall hardware footprint and can help reduce overall carbon footprint
- Better decoupling of hardware and software helps reduce lifecycle cost by minimizing application migration due to hardware obsolescence
- Virtual machines can rapidly be moved from one physical server to another, sometimes live with the applications still running. This dramatically reduces downtime during software updates and in the event of disaster recovery
- Centralized infrastructure simplifies the creation and management of common software loads, and facilitates application deployment and cybersecurity updates. It also simplifies operation and maintenance, especially backups.

A traditional flight training device often comprises multiple physical computers. Abstraction facilitates better optimization of these computers' physical resources, such as CPU, memory, and storage. Consequently, the various applications and services running on these physical computers can transform into virtual entities. These entities can then be deployed and executed independently of the underlying physical architecture.

Virtualization employs software to establish an abstraction layer above computer hardware. This layer facilitates the partitioning of a single computer's hardware resources – including processors, memory, and storage – into several virtual machines. Each VM operates with its own operating system (OS) and functions akin to a standalone computer, despite utilizing only a portion of the physical hardware resources of the underlying computer.

Figure 2. Traditional computing complex vs a data center approach using virtualization

Virtualization tackles hardware management and obsolescence issues but doesn't fully simplify complex simulation software. These models must accurately mirror complex aircraft systems, integrate with synthetic environments, and connect to instructor stations and courseware applications. The complexity of this interdependent system necessitates a solution for effective decoupling. The software industry's answer is a transition to a micro-service architecture, breaking a monolithic application into independent applications via a controlled API.

Containerization provides an ideal environment for these applications as it decouples the application from the infrastructure, packaging an application with its dependencies in an easy to deploy, “run-anywhere” bundle. This approach offers flexibility and portability, making it an attractive solution for managing simulation software complexities.

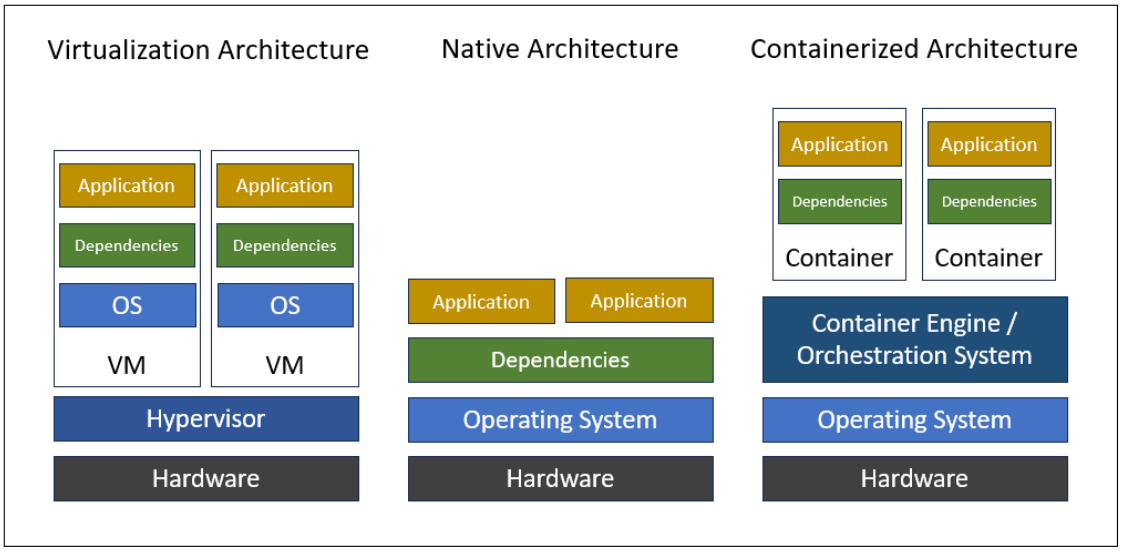

Figure 3. Comparison of native, virtualized and containerized architectures

As depicted in Figure 3, containerized applications require an operating system installed where the container engine operates. Containers are typically constructed by initiating a base container image and adding the necessary applications and their dependencies. The base image is often a streamlined version of an operating system. However, containers offer flexibility and do not necessarily require an OS-type base image. If an application can function without dependency on the OS, a container can be created from scratch, incorporating only the application executables and dependencies.

While containerization is a common practice in web-based applications, its application in flight training devices demands additional effort to run in a real-time environment. As a containerized application uses the Linux kernel of the host, it's possible in theory for such an application to run in real-time if the kernel allows it, in addition to directly using hardware devices if allowed.

The feasibility of running real-time applications within containers has already been explored in the industry, as evidenced by the work of (Kirkendoll, 2022). They found different areas of research (Fiori, 2022) (Hofer. 2021) (Struhár, 2021) that showed the possibility of achieving real-time for processes started through a container engine.

In the following sections, we will discuss the challenges associated with utilizing virtualization and containerization in a real-time environment.

VIRTUALIZATION CHALLENGES

The primary aim of abstraction is to enhance efficiency and optimize resource utilization by running multiple virtual applications on a single physical machine. Achieving real-time performance in virtualized environments can be challenging due to the potential overhead and latency introduced by the virtualization layer. Minimizing jitter and ensuring predictable response times necessitates meticulous configuration and tuning of both the virtual machines and the underlying hypervisor.

We used a popular virtualization environment, widely deployed in commercial data centers, for our tests. Initial analysis convinced us that it provided the necessary flexibility to allow for better control and predictable latency. The hypervisor, for example, offers numerous features that enable us to test the execution of systems requiring real-time performance.

The simulation processes running in each VM are scheduled using our dispatcher or simulator executive. It allows for scheduling of both synchronous and asynchronous processes. A synchronous process can be synchronized either through an external event or internally via the local timer. Synchronous processes operate at specific frequencies determined by the developer of the simulation package. For example, one civil aircraft simulation we have integrated is designed to operate at a frequency of 200 Hz.

Two factors are critical to the scheduling of real-time processes:

- Scheduling latency: the amount of time between the request to execute a process and the actual execution of the process. Scheduling latencies typically vary and can lead to uneven execution. For example, a process that is expected to repeat every 10 ms may in fact execute after 9.5 ms, then 10.5 ms, then 10 ms, and so on.

- Virtual machine synchronization: our architecture uses the data network to synchronize multiple computers (virtual or otherwise). A "network sync object" is used to synchronize the execution of multiple processes running on different computers. If there is a significant delay or a variation between computers, then this will further compound the uneven execution problem.

Scheduling

Scheduling latency can be improved by having only one real-time process bound to each CPU by setting the CPU affinity. In addition, our dispatcher provides a mode where a real-time process gets exclusive access to that CPU. In

a virtualized environment, this CPU assignment is applied to a virtual CPU and there is no exclusive physical CPU assignment, which could result in additional latency in the execution of a real-time process.

Conveniently, the hypervisor offers specific features focused on latency optimization. These features are designed to enhance the performance of virtual machines that require low-latency and real-time responsiveness. One of the key aspects of the latency-sensitive feature includes the CPU affinity. It assigns exclusively physical CPU cores to a VM to reduce latency by ensuring that the VM's processes run on dedicated cores without interruption from other VMs or host processes.

Enabling the latency-sensitive feature in the hypervisor has drawbacks however: (i) it can reduce scalability by limiting the host's resource-sharing ability among multiple VMs, leading to potential underutilization of hardware resources (ii) it often requires significant CPU and memory reservations, leaving fewer resources for other VMs (iii) the setup process is complex, involving careful configuration of various parameters (iv) prioritizing latency-sensitive VMs may increase latency or degrade performance in other VMs and (v) it may require more management effort to ensure proper configuration and maintenance of latency-sensitive VMs.

Synchronization

As stated earlier, executing the flight loop requires the synchronization of numerous tasks. These tasks may be spread across multiple virtual machines and physical computers. These tasks are carried out sequentially, with synchronization being facilitated via a sync object across the network. Therefore, the scheduling of this sync object to the various real-time processes is of paramount importance, as it directly influences the overall transport delay.

Several measures can be taken to minimize transport delay. One key approach is network traffic prioritization, which provides Quality of Service (QoS) at the Media Access Control (MAC) level. This allows the sync object to be given a high priority over less critical data. Such prioritization can be implemented at various levels, including network switches, network card drivers, and the network stack.

In a virtualized environment, the network layer is abstracted using virtual switches and virtual adapters, where traffic prioritization might not be inherently applied. Additionally, there is no direct control over the execution of network I/O, potentially causing latency that directly affects real-time process scheduling. One way to minimize potential latency is to ensure that physical CPU resources are always available to process network I/O at the hypervisor level. As we have seen, this can lead to underutilization of the hardware resources.

Another option that was evaluated was the SR-IOV[3] technology, which allows a single physical network interface card (NIC) to be shared efficiently among multiple virtual machines. This approach minimizes virtualization overhead at the hypervisor level. However, this technology introduces complexity in configuration and management and restricts the use of advanced hypervisor features such as dynamically moving VMs on available hosts.

Table 1. Categories of virtual machines

VM Categories	Description
1- Sync provider	At least one real-time process in the VM generates a network sync object
2- External sync-ed	At least one real-time process in the VM is scheduled on the reception of the network sync object
3- Local scheduling	All the real-time processes on the VM are scheduled using the local timer
4- Management	This category is used for VMs that operate and manage the simulation and its environment and contain no real-time processes.

[3] The Single Root I/O Virtualization (SR-IOV) specification was developed by the PCI-SIG (Peripheral Component Interconnect Special Interest Group).

Given these limitations, an alternative approach was explored. Since only certain processes require synchronization, it is crucial to classify the VM profiles and assign them to the appropriate servers. This targeted allocation helps optimize performance and manage resources more effectively. Table 1 outlines different VM categories and their corresponding profiles.

As indicated by the table, only VMs in categories 1 and 2 require synchronization. Consequently, the latency-sensitive performance setting is applied only to those categories. Additionally, servers hosting VMs in those categories must be under-provisioned by allowing enough spare processing to ensure that physical CPUs are always available to process network I/O. By restricting this setting to categories 1 and 2, the previously mentioned drawbacks of latency-sensitive are minimized.

Containerization

As previously discussed, containerized applications use the host's kernel and it's therefore important that the computer that will run the container engine will also be properly configured for real-time.

The kernel options typically required for full flight simulator real-time computers include:

- Scheduling: the SCHED_FIFO scheduling option guarantees that processes with higher priority are executed first and without any disruption.

- Preemption: preemption in the kernel minimizes the scheduling delay of high-priority tasks, which is crucial for real-time computing. The CONFIG_PREEMPT_RT patch can make nearly all parts of the kernel preemptible, enhancing responsiveness.

- Priority: to utilize the above kernel settings, real-time processes must be started with low "niceness" and high priority allowing them to preempt non-real-time processes, and ensuring they receive CPU time when needed.

Containerization is primarily used for web applications, and as a result, the default configurations of most container engines are optimized for this domain rather than for real-time applications. Engines like Docker offer a flexible level of abstraction between the application running in the container and the host's operating system. Therefore, efforts need to be invested in understanding these layers to strike an optimal balance between performance and isolation.

In our exploration of containerization, we focused on the technical possibility of running part of existing full flight simulator software in a real-time environment. To keep existing application and hardware architecture untouched as much as possible we had to relax some aspects of containerization isolation aspects.

Additional Considerations

Implementing abstraction on full flight simulator is an important technical challenge. However, it is not the only challenge and while this paper cannot go in details for every challenge, we present some of them here for the consideration of the reader. Some of them were nonetheless tackled as part of our virtualization and containerization effort.

- Cost and licenses: while higher density of application lowers the total amount of resources like memory and CPU required this could be offset by the license cost of some commercial abstraction software

- New operations procedures: as with any new technology the change management aspect should not be neglected. For example, a virtualized cabinet maintenance procedure like backup and disaster recovery is different that classical cabinet and will require training.

- Third party software support: while many applications will run in an abstracted environment without issue, some applications are tightly coupled to hardware devices and will require refactoring. This is an important issue when the affected software comes from a third party, for example from an aircraft original equipment manufacturer (OEM) with their own roadmap and priorities.

- Refactoring effort: at the outset of any virtualization project, it's important to account for the effort required to refactor applications to ensure they run correctly. The resulting analysis may show that while a solution may be technically possible, it may not be commercially viable if the effort required to implement it is too high.

TEST SETUP AND RESULTS

In real-time processing, our primary concern is minimizing the worst-case latency. This is more critical than reducing the average latency. The simulation processes are considered as time-critical, where some variation on the timing can be tolerated as long as it's within defined deadlines[4]. The impact of missing deadlines depends on their frequency and severity. Occasionally missed deadlines won't cause a complete simulation failure, but they can degrade performance, potentially affecting the user experience (e.g., pilot display glitches or user/instructor station interactions). Significant performance issues could ultimately result in motion shutdown or aircraft hardware failure (if actual aircraft hardware is used) when the system detects abnormal behavior.

To evaluate the impact and effectiveness of various tests, we employed a comprehensive set of performance indicators. This study will concentrate on two metrics that aptly reflect the simulator's overall performance. The first metric pertains to the accuracy of scheduling simulation iterations. The second metric deals with the duration needed to complete a full flight loop.

Testing was performed on three different simulators representing three different commercial aircraft. The results for two of those aircraft representing two different base frame rates are presented herein.

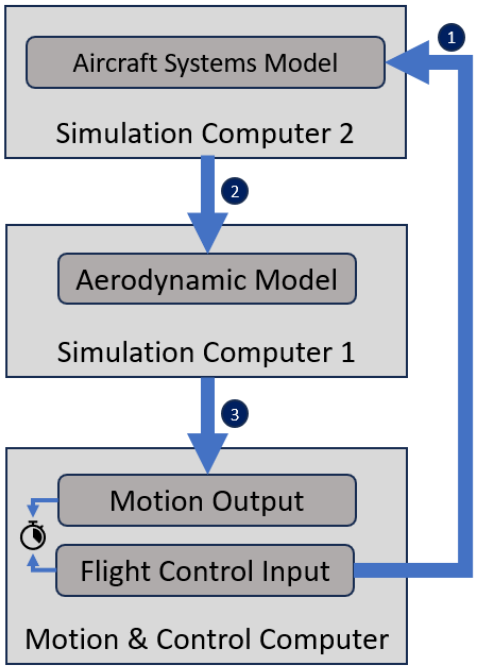

Figure 4. Simplified simulation path. The numbered arrows show the sequence of network data transfers that occur each simulation frame. Note that in our architecture, the motion and flight control simulation is run on the same computer, so the sequence starts and ends at the bottom of the diagram.

Simulation Scheduling

The scheduling of a process is measured to assess real-time performance of the virtualized environment. This measurement is obtained by calculating the time variation between two iterations. The deviation from the scheduled time is referred to as jitter, emphasizing the need for a measurement tool to evaluate real-time performance. Our team developed a software tool that timestamps the beginning of each iteration in real-time. The collected data can then be analyzed. This tool is crucial as it provides a means to compare and validate performance while experimenting with different scenarios or settings.

The time required to do a flight loop, identified as transport delay, is the time it takes for an input from the pilot to be seen or felt by the pilot. Part of this loop includes systems that were not affected by this study, for example the projector delay of the visual system. We call simulation delay the signal path that starts at the detection of the flight controls movement by the simulation to the output of the signal to the motion system. This is an important metric as this is the part of the transport delay that is running though the computers affected by virtualization and containerization. A diagram of this is shown on Figure 4.

Figure 5 illustrates the performance scheduling of the main simulation process that operates at an execution frequency of 100 Hz. The graphs depict the scheduling jitter, which is the time elapsed since the previous scheduling. Each graph represents approximately five minutes of data collection.

[4] The computer driving the motion and control loading system is a notable exception, running a dedicated hard real time operating system and not a target for virtualization at this time.

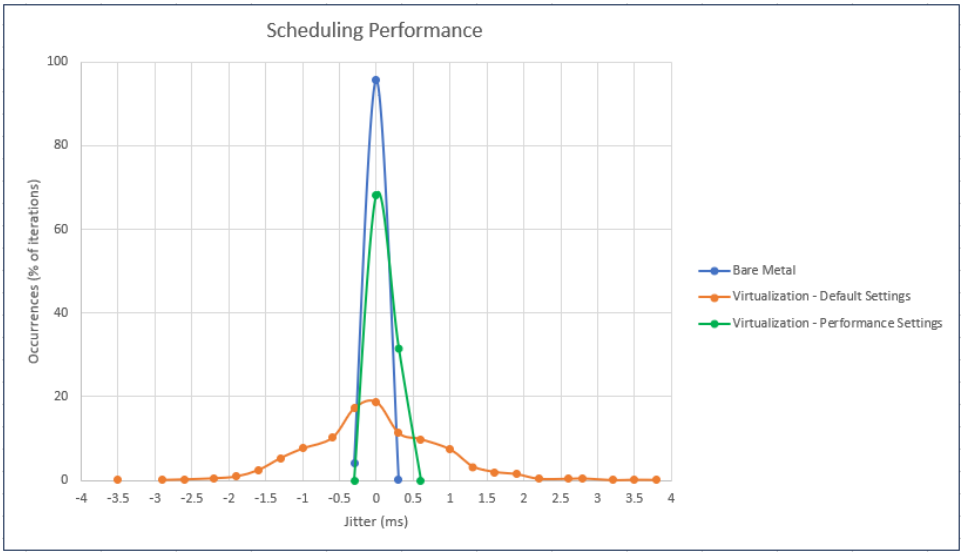

Figure 5. Comparison of real-time scheduling performance on simulator #1 using multiple virtualization settings and bare metal. The narrower the peak of results, the better the performance. The base frame rate for this simulator is 100Hz.

The first graph represents data collected from a bare-metal computer complex, serving as our reference for comparison with virtualization. The vertical axis indicates the percentage of occurrences for each time interval (bucket) on the horizontal axis. The second and third graphs show the simulation running in a virtualized environment, with both using the same VM allocation on each server. As seen in the second graph where the default virtualization settings were used, there is significantly more jitter. For example, nearly 10% of all simulation frames started either 1ms early or 1ms late. The third graph shows the results after applying latency-sensitive performance settings, resulting in performance scheduling that closely matches the bare-metal reference.

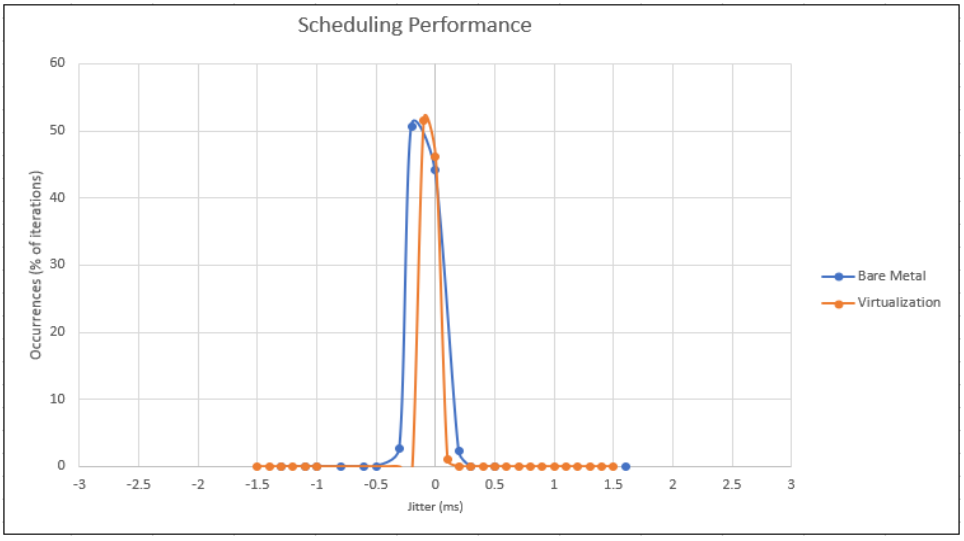

Figure 6. Comparison of real-time scheduling performance on simulator #2 using virtualization and bare metal. The narrower the peak of results, the better the performance. The base frame rate for this simulator is 200Hz.

In these tests, category 1 and 2 VMs (refer to table 1) are hosted on an under-provisioned server to ensure available physical CPU resources for performing I/O networking. The other VMs are hosted on an over-provisioned server. As

mentioned earlier, this solution has some drawbacks: it underutilizes the benefits of virtualization and adds complexity in managing VM allocation on each server. Ideally, such manual management of VM allocation would be unnecessary. For instance, if the hypervisor could provide control of I/O networking and support QoS priorities, it could help achieve an optimal solution.

As part of our testing to determine the best performance with virtualization, we also experimented with selecting VMs and their CPU socket architecture by aligning it with physical hardware (NUMA nodes) to maximize performance. In addition, we modified CPU share priority to manage and prioritize the allocation of CPU resources among VMs based on their relative importance. All the results in this paper benefitted from these settings as well.

Figure 6 shows the results for simulator #2, comparing virtualization with bare metal. Data was collected over a period of forty minutes and the base frame rate of this simulator is 200Hz. Once again, the virtualization results compare favorably with the traditional bare-metal implementation. Along with the (unpublished) results from simulator #3, these positive findings enabled us to move on to the testing and measurement of the overall simulation and transport delay. As previously mentioned, transport delay is crucial for qualifying a simulator for training by civil aviation authorities.

Simulation Delay

Simulation delay, as shown in figure 1, was tested over a 15 minute period with the average and maximum delay being logged. The results are summarized in table 2. The maximum time observed is the more important metric and as can be seen from the results, they compare favorably with our reference bare metal implementation.

Table 2. Comparison of results for simulation delay between bare metal and virtualization

Bare metal		**Virtualized**	
Average (ms)	Max (ms)	Average (ms)	Max (ms)
13.0	15.0	12.5	15.0

Beyond the testing just described, we performed additional objective and subjective tests. The objective tests were the ones required by civil aviation authorities for simulator qualification. As a final check, pilots executed multiple training sessions to subjectively evaluate performance; they did not notice a difference.

Our first simulator implementing this virtualized architecture is currently operational and in training, qualified under ICAO Level 7, which is equivalent to FAA/EASA Level D.

Containerization

In parallel with the evolution of virtualization, we experimented with running portions of a full flight simulator's software as containerized applications. Our initial goal was to execute two Linux applications full flight simulator within containers. These applications, integral to the flight loop, are highly sensitive to synchronization jitters and invariably require the real-time features of the Linux kernel.

This effort highlighted the importance of having independent modules to fully capitalize on containerization. The two applications communicate via shared memory for data exchange and synchronization. This served as an opportunity to investigate if we could containerize legacy applications without refactoring.

We utilized Docker as our container engine, placing the main executable of each application into individual containers, complete with all necessary dependencies. The container images were based on the RedHat Universal Base Image to align with the current simulator operating system. A remaining research area is to explore the potential for maximum application isolation by starting with an empty (referred to as 'scratch') base image. This approach could result in a cleaner image that is easier to maintain and deploy, albeit potentially more challenging to debug.

Some of the highlights of our testing included:

- Process Priority: processes running inside containers are subject to limitations imposed by the container engine. We changed the docker engine parameter to allows processes within the containers to run in real-time priority.

- Shared Memory Access: for our proof-of-concept, the shared memory used for communication between the applications was created on the host machine, and containers were allowed to access the host shared memory directly.

- Network Adapter Access: synchronization events between various processes are done using a sync object over the network. To ensure minimal delay between the reception of the sync object by the host and the interrupt in the application, containers were permitted to connect directly to the hardware (host) network adapter.

This implementation enabled us to achieve our goal of running two critical containerized real-time applications on a full flight simulator without any performance degradation. Our primary performance metric was the simulation delay, which is, as mentioned previously, the subset of the transport delay that is impacted by a move to the containerization. Our measurements indicated that the simulation delay remained consistent, irrespective of whether the applications were containerized, as shown in figure 7.

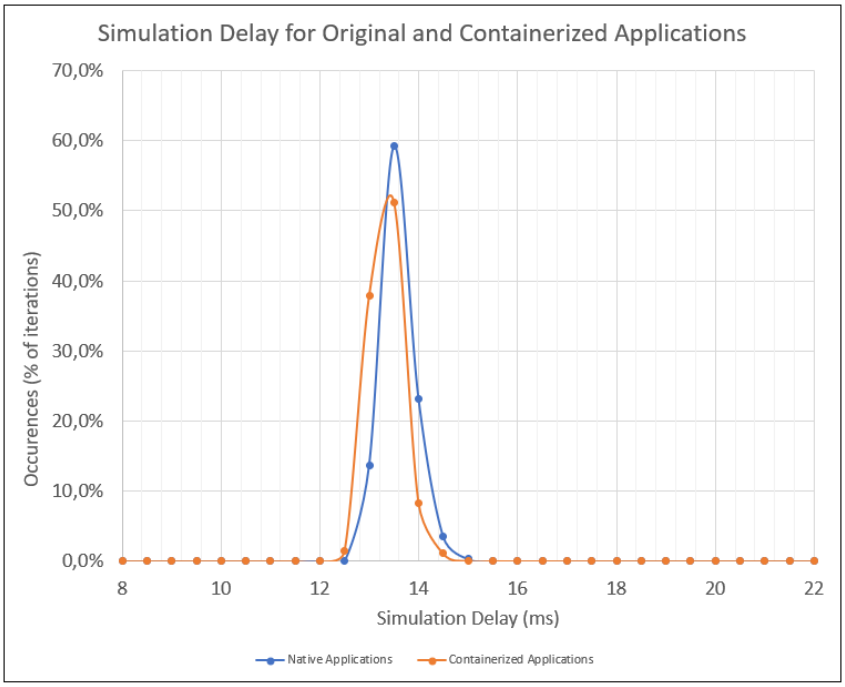

Figure 7. Simulation delay of native and containerized applications

As can be seen, the simulation delay compares favorably with the native implementation. To further validate the results, we repeated the objective and subjective qualification tests, with no issues encountered. This gives us the confidence to pursue containerization on a full flight simulator in a real time environment, either to replace or as a complement to virtualization.

CONCLUSION AND RECOMMENDATIONS

To fully harness the flexibility offered by virtualization and containerization, it's crucial for the industry to adapt the development of flight training device software to align with the design paradigm of these underlying technologies. To maximize the benefits of virtualization, we need to transition from monolithic applications with large computational requirements to multiple smaller, more manageable applications. This shift allows for a more flexible and efficient allocation of virtual machines on their physical host.

Considering that low latency applications require a special setup for use in virtual machines, it's equally important to establish clear separation of concerns. Applications requiring low latency should be isolated in a virtual machine, with other, less sensitive aspects of the simulation running on less restrictive virtual machines.

Containerization takes this paradigm shift a step further. We envision simulators fully leveraging containerization to have each aspect of the simulation isolated as an independent application deployed as containers. This approach promotes better flexibility and optimizes the usage of available hardware resources.

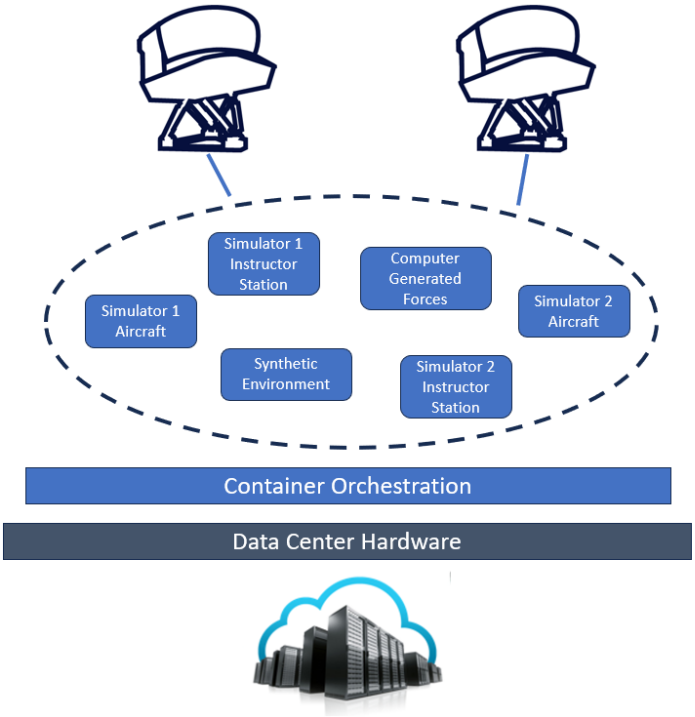

Figure 8. Hypothetical architecture of a modular training device ecosystem

To achieve a common deployment technique with clear interfaces between modules, we will need to review the design of different modules to ensure their independence and well-defined interfaces. This approach will enhance collaboration between teams, with resources spending more time developing applications and less time on integration.

REFERENCES

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. IEEE. 10.1109/ISPASS.2015.7095802

Fiori, S., Abeni, L., & Cucinotta, T. (2022). RT-Kubernetes—Containerized Real-Time Cloud Computing, SAC '22: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. 10.1145/3477314.3507216

Hofer, F., Sehr, M., Sangiovanni-Vincentelli, A., & Russo, B. (2021). Industrial control via application containers: Maintaining determinism in IAAS. Systems Engineering. 10.1002/sys.21590

International Civil Aviation Organization – ICAO (2015). Manual of Criteria for the Qualification of Flight Simulation Training Devices, 4th Edition, Volume 1 Airplanes, Part II Flight Simulation Training Device Criteria

Kirkemdoll, Z., Lueck, M., (2022). Real-time Simulation Executive Architecture and Subsystem Containerization, Interservice/Industry Training, Simulation, and Education Conference, 22403

Knobbout, J., (2023). Virtualized Simulation for Military Concept Development and Experimentation: The Cerebro Battle Lab, a Case Study, Interservice/Industry Training, Simulation, and Education Conference, 23372

Moy, M., Fowler, D., (2023). Reducing Image Generator Footprint with Virtualization, Interservice/Industry Training, Simulation, and Education Conference, 22438

Struhár, V., Craciunas, S. S., Ashjaei, M., Behnam, M., & Papadopoulos, A. V. (2021). REACT: Enabling Real-Time Container Orchestration. IEEE International Conference on Emerging Technologies and Factory Automation. 10.1109/ETFA45728.2021.9613685