HYPERVISOR EVALUATION FOR VIRTUALIZATION OF A HIGH-PERFORMANCE SMALL SATELLITE PAYLOAD

Maria Jose Luna Mejia, Konstantin Schäfer, Christian Heim, Clemens Horch, Frank Schäfer, Stefan Rupitsch

Fraunhofer Institute for High-Speed Dynamics, Ernst-Mach-Institut EMI

Ernst-Zermelo-Straße 4, 79104 Freiburg im Breisgau, Germany

Abstract

The emerging field of NewSpace technologies has led to the development of cost-efficient small satellites by utilizing commercial-off-the-shelf (COTS) components in their designs. Small satellites face significant challenges due to their limited power resources and exposure to harsh environmental conditions, like radiation or temperature fluctuations. These can cause hardware and software failures that compromise the reliability of the satellite operations. This paper focuses on investigating the performance impact of virtualization in the context of COTS-based small satellite payloads. It specifically explores the application of the open-source virtualization solution Xen and its potential to enhance the software reliability of small satellite payloads based on a multiprocessor system-on-a-chip (MPSoC). The study evaluates how virtualization enables the strict separation of mission-critical and non-critical software modules through the use of virtual machines, ensuring isolation. Through experiments conducted on the ZynqMP UltraScale+ MPSoC-based data processing unit (DPU) developed at Fraunhofer EMI for small satellite payloads, the results demonstrate a significant improvement in the execution time and variance of a real-time application when executed in a separated, virtualized environment. This observation highlights the value of a virtualization setup with CPU pinning, which enhances predictability, especially under high system loads. Furthermore, leveraging virtualization in small satellite systems ensures strict isolation between applications with different confidence levels, enabling multiple users to access and utilize the same hardware platform. This approach also facilitates satellite-as-a-service models, promoting cost sharing, collaboration, and reducing barriers for small satellite operators to provide shared infrastructure or engage in joint missions with other institutions.

Keywords: NewSpace, virtualization, small satellite, reliability

1. INTRODUCTION

In recent years, the development of small satellites has gained significant interest due to their cost-efficiency and flexibility. Many small satellites utilize commercial-off-the-shelf (COTS) components to reduce development time and launch costs. However, the use of COTS hardware in space evokes challenges in regard to reliability and lifespan, mainly due to the extreme environmental conditions such as temperature changes and radiation. These factors can lead to hardware and software failures, impacting the reliability of small satellite operations. Additionally, some small satellites have experienced mission failures due to limited development budgets and compressed timelines, which often involve accepting certain risks. Unfortunately, the flight software (FSW) of small satellites is often neglected until the late stage of spacecraft integration, within an already constrained development cycle. Consequently, FSW testing can be inadequate, potentially leading to undetected software errors or, at worst, mission failure [1]. In contrast, larger missions with more extensive resources may employ thorough testing, including simulations, to enhance FSW reliability. This way, comprehensive FSW coverage can be achieved on those missions and the software is expected to operate without catastrophic failures once deployed in orbit [2].

To address described software failures in small satellites, this paper proposes the utilization of virtualization techniques. Virtualization in the context of space missions is an actively explored domain. This evolving technology enables the execution of multiple virtual machines (VMs) on a single physical machine, which holds considerable benefits for resource-constrained space missions that require high reliability and flexibility. By implementing virtualization in small satellites, it becomes possible to combine multiple subsystem functions on a single processing system encapsulated in individual VMs, resulting in reduced satellite size, weight, and power requirements. Furthermore, virtualization facilitates software isolation among different applications running concurrently on the same device, ensuring that failures in one application or VM do not adversely impact others or the overall system. In this paper, we assess the potential of virtualization in enhancing the reliability and performance of small satellite payloads by evaluating the Xen hypervisor. The evaluation of the Xen hypervisor is performed on a ZCU104 evaluation board from Xilinx, which is equipped with a Zynq UltraScale+ Multiprocessor System-on-a-Chip (MPSoC). This MPSoC is similar to the one used in a Data Processing Unit (DPU) for small satellite missions developed by Fraunhofer EMI [3].

The chosen test setup simulates operations typically executed by small satellites. This involves running a real-time (RT) application in parallel with stress tests designed to mimic resource-intensive or rogue on-board processing tasks under various operating system scenarios. This facilitates the evaluation of how virtualization impacts both the performance of the RT application and the overall system.

By running a high-priority RT application in one VM and executing stress tasks in another VM, it could be shown that the performance of one application does not noticeably interfere with the operation of the other. These insights highlight the potential of virtualization techniques in improving the reliability and performance of small satellite systems. Furthermore, virtualization can be explored as a NewSpace solution, enabling collaboration between institutions and cost-sharing through the shared use of satellite hardware.

2. RELATED WORK

The field of safety-critical hypervisors in the aerospace domain has seen advancements in recent years, with hypervisors like XtratuM [4]. As a type-1 (bare-metal) hypervisor, XtratuM utilizes para-virtualization techniques and adheres closely to the ARINC 653 standard [4], making it suitable for safety-critical aerospace applications. XtratuM has demonstrated its versatility by successfully being ported to reference platforms within the spatial sector, including LEON2 and LEON3. These implementations have further validated its capabilities and suitability for safety-critical aerospace applications. Moreover, in line with the growing demand for multicore platforms, the MultiPARTES project [5] has extended XtratuM to provide support for multicore environments, enhancing its scalability and performance [6].

Additionally, Steven VanderLeest developed an early prototype of an ARINC 653 implementation using the open-source Xen hypervisor and a Linux-based domain/partition operating system. This prototype, known as ARLX (ARINC 653 Real-time Linux on Xen) [7], was later enhanced with additional safety and security features. A notable aspect of ARLX was its innovative approach to multicore processors. In a more recent work [8], VanderLeest's proposal focused on secure virtualization deployed on heterogeneous multicore platforms (HMP) [9].

3. MATERIALS

In light of these developments in the field, this paper aims to evaluate the efficacy of virtualization, specifically the Xen hypervisor, in enhancing the dependability and efficiency of small satellite payloads within space environments.

3.1. Xen hypervisor

Xen is an open-source type-1 hypervisor that enables virtualization of hardware resources. It provides an abstraction layer between the physical hardware and virtual machines, offering features such as virtual machine creation, resource management, as well as network and storage connectivity [10]. According to the Xen hypervisor terminology, the main source or virtual machine monitor is Domain-0 (Dom0). It is a privileged virtual machine that provides access to the management and control interface to the hypervisor itself. Standard guest operating systems exist in unprivileged domains, known as DomU and they can run different guest software stacks, like operating systems.

The Xen hypervisor offers several practical benefits that make it a strong choice for various applications, especially in the context of satellite systems. Firstly, Xen's widespread use in data centers speaks to its reliability and robustness. It has been extensively tested and proven in real-world scenarios, which is crucial for applications where downtime or failures can have serious consequences. Additionally, Xen is compatible with different architectures, particularly embedded systems and it is open source. This means that it can be customized and modified to meet specific requirements, which is particularly valuable for satellite missions that often have unique needs. Furthermore, collaboration and cost-sharing become feasible in a satellite system by hosting multiple virtualized missions on a single platform, ultimately making space exploration more affordable and accessible. Maintenance and upgrades are simplified, with individual virtual machines (VMs) easily updated without affecting the overall system.

3.2. Zynq UltraScale+ MPSoC ZCU104

As an evaluation platform, the Xilinx ZCU104 development board with a Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC was selected. This platform was chosen to execute the tests over the actual Fraunhofer EMI's DPU since it is built to run as a standalone device and therefore has a standard power supply and provides easily accessible interfaces.

4. METHODS

This paper undertakes a study involving the parallel execution of two applications within different testing environments. The applications in focus are a RT application, commonly employed in small satellites and a stress-inducing command intended to impose a substantial load on the MPSoC. The testing environments are the following:

- A standard GNU/Linux environment
- A GNU/Linux environment with different scheduling priority for each task
- A Xen hypervisor utilizing one virtual machine for the RT application and another for the stress command, both running a GNU/Linux operating system
- A Xen hypervisor alike the previous scenario but incorporating CPU pinning for each VM.

4.1. Description of applications

To replicate the DPU's processing capabilities within the context of a small satellite, a RT application was developed, which is frequently utilized in similar satellite systems. This application operates in real-time by delivering instantaneous results and data updates upon command execution. As such, the RT application serves as a suitable means to assess the DPU's capacity for managing and processing data in real-time scenarios.

As a secondary application, the "stress" command is used. It is a powerful utility commonly used in Unix-like environments to impose a substantial load on a computer system. Its primary purpose is to facilitate testing and evaluation of system performance and stability under high-stress conditions. By specifying various parameters, users can control the type and intensity of the workload generated by the stress command [11]. This paper focuses on evaluating the “cpu”, “vm” and “io” stress options. Once initiated, the "stress" command generates the specified workload, causing the system to heavily consume CPU, memory, or I/O resources respectively.

4.2. Testing environments

In the context of the paper, the GNU/Linux image refers to the baseline operating system (OS) currently running on the DPU. In this testing environment, the RT application and the stress command are executed concurrently within the GNU/Linux OS. This setup allows for the evaluation of the system's performance and behavior under normal operating conditions, serving as a reference point for comparison with the subsequent testing scenarios.

The second testing scenario is the GNU/Linux image with scheduling priority. Linux scheduling is a fundamental aspect of the GNU/Linux operating system, determining the execution order of processes and threads. The scheduler continuously monitors and adjusts the placement of processes on the run-queue and CPU, considering factors such as CPU utilization and priority [12]. By assigning higher priority to the RT application while running concurrently with the stress command, this testing scenario explores the impact of prioritization on the system behavior and resource allocation. This is done in order to evaluate how Linux scheduling influences the performance and responsiveness of the system, particularly in the presence of resource-intensive tasks.

In the Xen testing scenario, the environment is set up with two virtual machines. The first VM runs the RT application, while the second VM executes the stress command. This configuration ensures that the resource consumption of the stress command does not impact the performance of the RT application, as they are isolated within separate VMs. By utilizing virtualization, this testing scenario aims to evaluate the effectiveness of maintaining the RT application’s reliability and performance.

In the Xen image with CPU pinning testing scenario, the assignment of CPU resources to the virtual machines is explored. The VM running the RT application is assigned to CPU 1, while the VM running the stress command is allocated CPU 2 and 3, leaving CPU 0 available for Dom0 or the privileged VM. A representation of this configuration is shown in Figure 1. This allocation is achieved through CPU pinning, a mechanism provided by Xen that restricts the utilization of virtual CPUs to specific groups of physical CPUs. By employing this technique, the precise allocation of CPU resources is ensured, preventing the VM running the stress command from monopolizing all available CPUs and potentially impacting the performance of the VM running the RT application. Additionally, CPU pinning guarantees that each VM has dedicated access to its allocated resources, enhancing isolation and maintaining consistent performance [13].

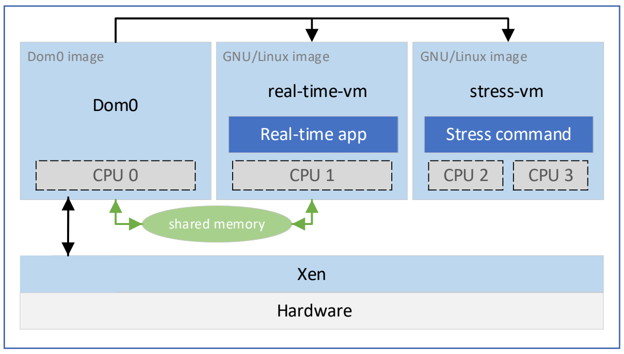

FIGURE 1. Representation of Xen image with virtual machines and CPU pinning.

5. IMPLEMENTATION

In order to assess the real-time capabilities of the RT application, its execution time (Δt) is measured using GPIOs of the ZCU104 board and an oscilloscope. The execution is triggered when a rising edge is detected at GPIO1, and upon completion, it toggles GPIO2. Δt is calculated by measuring the time between the rising edge of GPIO1 (when its voltage exceeds a threshold of 2.5 V, indicating the transition from 0 to 1) and the rising or falling edge of channel 2 (when its voltage surpasses or drops below 2.5 V, indicating the toggling of GPIO2 after executing the real-time application). Figure 2 illustrates the voltage levels of channel 1 and 2 during the execution of the real-time application, resulting in a Δt value of 0.7 ms.

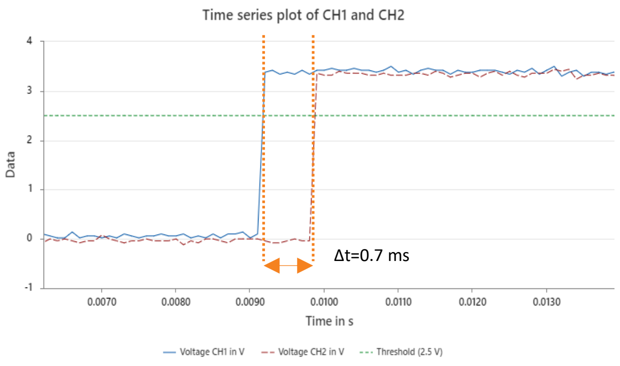

FIGURE 2. Exemplary plot of data obtained from oscilloscope and resulting Δt.

The performance evaluation of the RT application and the system is conducted in this paper under the different testing environments and stress configurations. The testing involves examining various combinations of stress commands, such as different options of the "cpu" "vm" and "io" workers. In total, 35 different stress combinations are evaluated to assess the performance of the system and the RT application for each testing scenario.

To assess the real-time application's execution time under different stress conditions, the application was executed 500 times in each subtest (representing a specific combination of “cpu”, “vm” and “io” stress workers) within each scenario (GNU/Linux, GNU/Linux+priority, Xen, and

Xen+CPUpinning). This methodology allowed for the collection of 500 data points (Δt) for each subtest within each scenario, enabling the calculation of the mean and standard deviation of Δt. The execution of 500 RT application runs in each subtest constituted a batch, resulting in a total of four batches for each scenario. This yielded a comprehensive data set of 2000 data points per scenario, ensuring statistical significance in the analysis of performance metrics across different stress combinations. Furthermore, conducting the tests in four separate batches, rather than gathering all 2000 data points consecutively, helps mitigate setup errors and minimizes the impact of external factors that may vary over time. This approach ensures more reliable and consistent results throughout the evaluation process.

Alongside measuring the execution time of the RT application, other system metrics were also recorded, including the load average, memory usage, and stress bogus operations. These metrics provided valuable insights into the system's behavior under stress and facilitated the identification of potential bottlenecks and areas for optimization.

5.1. Mean and standard deviation of execution time of real-time application

The mean execution time represents the average or expected performance of the application under a specific scenario, while the standard deviation indicates the degree of variability or inconsistency in its performance. By examining both the mean and standard deviation of the execution time, we gain a deeper understanding of how the real-time application behaves in different scenarios and potentially identify factors that contribute to performance variations. These metrics also allow to track variations in execution time across environments and evaluate the RT application's stability.

5.2. Load average

The load in a GNU/Linux system is a metric that indicates the current CPU utilization by measuring the number of processes being executed or waiting for execution. A load of 0 represents an idle system, while each additional process in execution or waiting increments the load by 1. On a quad-core processor, a load of 1 corresponds to 25% CPU usage, while a load of 4 corresponds to 100% CPU usage. However, the instantaneous load value alone does not provide meaningful information due to its rapid fluctuations. Therefore, the load average over a specified time period is used to monitor resource utilization [14].

To analyze the load in each subtest, the load average at the end of each subtest is monitored. Specifically, we focus on the 15-minute average load value as it offers a stable and representative measure of the CPU's average load throughout the entire subtest, which lasts approximately 21 minutes. By evaluating the load average over this duration, it gives valuable information about the resource utilization patterns during the subtest and can assess the system's performance in managing the workload.

5.3. Memory usage

Monitoring the memory usage is important during stress testing to ensure consistent system performance. Stress tests can consume system resources significantly. Tracking changes in used memory allows us to draw conclusions regarding resource utilization and potential bottlenecks. As the number of “vm” stress workers increases, the system's memory requirements also increase. By logging available memory values every 2 seconds during each subtest and capturing total memory data once per subtest (as it remains constant), potential memory limitations that may impact system performance can be identified. Additionally, to facilitate comparisons across subtests and testing scenarios, the area under the curve of used memory versus the duration of the subtest is calculated, providing a quantitative measure for evaluating memory utilization as it can be seen in Figure 3.

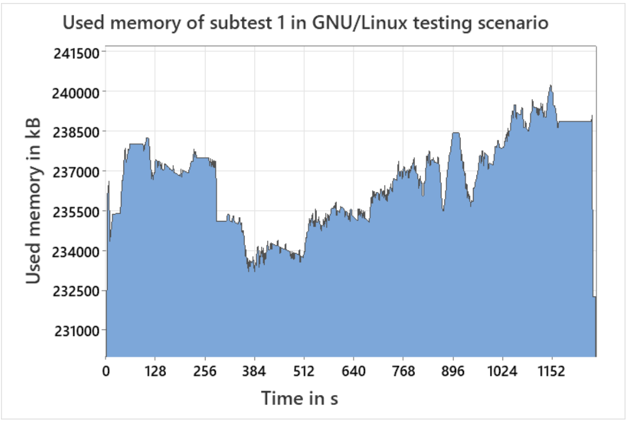

FIGURE 3. Plot of used memory vs. time during a subtest.

5.4. Stress bogus operations

The stress command utilizes "bogus operations per second" [bogo/s] as a metric to measure the amount of stress imposed on the system. During a stress test, various workloads are generated to stress different system aspects. The bogus operations represent a type of load that exerts stress by performing calculations that do not yield any useful output. Measuring the rate at which the system executes these bogus operations allows stress to estimate the overall processing power of the system under load. A higher number of bogus operations per second indicates a higher level of stress on the system [15]. The specific number of bogus operations varies depending on the stressor being executed. While it is acknowledged that bogus operations per second may not serve as a universally viable benchmark for system performance, it is important to note that in our specific use case of relative comparison, they can provide insights into the performance differences among the different testing scenarios.

6. RESULTS AND DISCUSSION

The boxplots shown in Figure 4 and 5 show the mean and standard deviation of the execution time of the RT application across the different stress combinations and testing environments. In the analysis, the interquartile range (IQR) provides information on the range of values that the middle 50% of the data falls into. The GNU/Linux environment exhibits the highest values for IQR and median for both the mean and standard deviation of the Δt of the real-time application. On the other hand, the Xen+CPUpinning environment shows the smallest IQR and median for both the mean and standard deviation of Δt, indicating a smaller range of execution time values. Notably, the Xen+CPUpinning environment displays the least disparity and lowest mean execution time compared to all other testing environments. This observation implies that the Xen+CPUpinning setup holds the potential to provide a more consistent and predictable performance for the real-time application.

The analysis reveals significant variations in the mean and standard deviation of the RT application's execution times across different testing scenarios. Specifically, the Xen+CPUpinning scenario exhibits the smallest statistical values compared to the other scenarios, with a mean Δt of 0.7505 ms and a standard deviation of 0.6820 ms. In contrast, the GNU/Linux scenario displays the highest statistical values, with a mean Δt of 2.757 ms and a standard deviation of 3.781 ms.

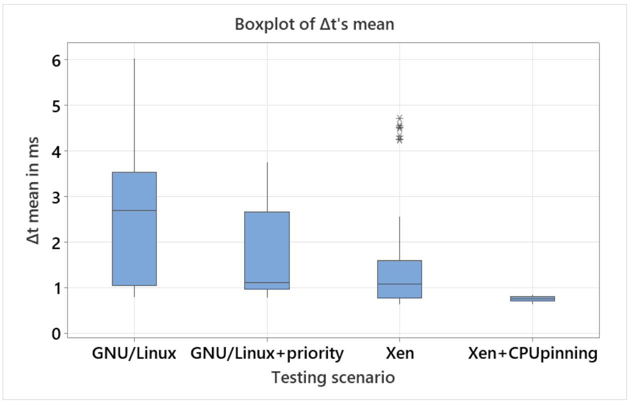

FIGURE 4. Boxplot of Δt's mean of the different testing scenarios.

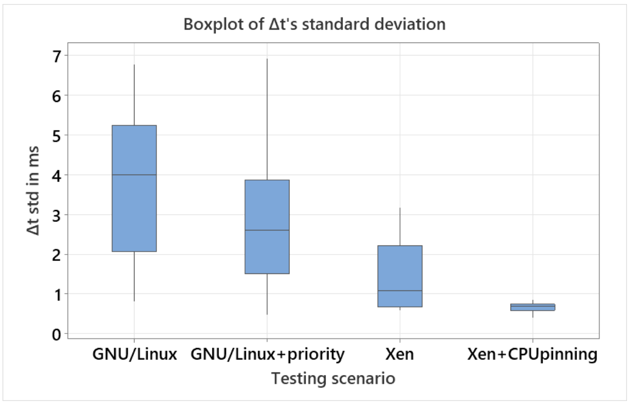

FIGURE 5. Boxplot of Δt's standard deviation of the different testing scenarios.

When analyzing the average load results, Figure 6 presents the aggregate average load of different stress combinations for each testing scenario. The graph showcases the sum of the 35 average CPU load values within each scenario. The non-virtualized environments GNU/Linux and GNU/Linux+priority, exhibit an identical total system load of 140.33. In contrast, the virtualized environments, Xen and Xen+CPUpinning, demonstrate higher total system loads of 144.04 and 144.73, respectively. This indicates a 2.64% increase in average load from the non-virtualized scenarios to Xen, and a 3.13% increase from the non-virtualized scenarios to Xen+CPUpinning. These findings suggest that virtualization introduces additional overhead, resulting in a higher overall system load compared to non-virtualized environments.

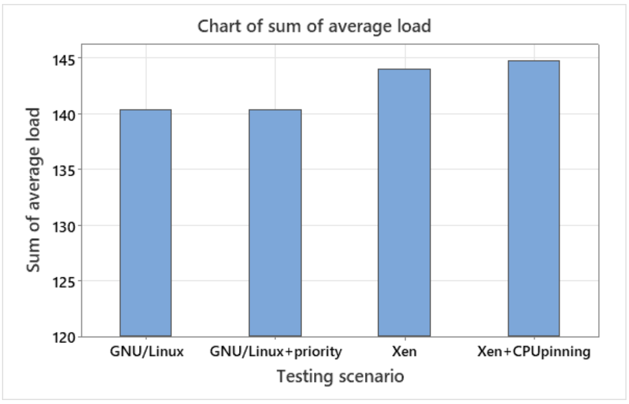

FIGURE 6. Sum of average load per testing scenario.

When examining the results related to used memory, a distinction can be observed between the subtests with zero and non-zero "vm" stress workers. This differentiation is made since the "vm" stressor directly impacts memory utilization. To analyze these results, the boxplot presented in Figure 7 is utilized. Notably, the boxplot for the Xen+CPUpinning scenario displays a narrow IQR for non-zero “vm” workers. This suggests that the used memory values in this scenario are tightly clustered around the median, indicating consistent memory resource management. One possible explanation is that the utilization of CPU pinning in the Xen hypervisor enables improved memory management and allocation, resulting in reduced variability in used memory values. Additionally, it is worth noting that the mean used memory is slightly higher compared to the Xen scenario, exhibiting a difference of approximately 2-3%. Further investigation could be conducted to explore the underlying cause of this increased memory usage and its implications.

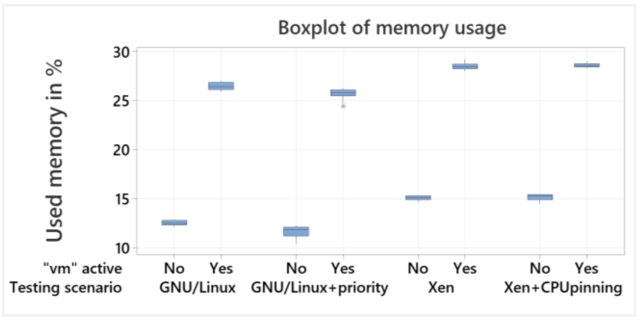

FIGURE 7. Boxplot of used memory in testing scenarios, considering zero and non-zero “vm” workers.

The cumulative number of bogus operations performed across all four testing scenarios is presented in Figure 8, offering a comparative analysis of this metric among the different scenarios. The results reveal that the GNU/Linux scenario demonstrates the highest sum of bogus operations in all stress tests, with a total of 1,742,887 operations. It is followed by the GNU/Linux+priority scenario with 1,672,763 operations, the Xen scenario with 1,205,589 operations, and finally, the Xen+CPUpinning scenario with 837,020 operations. Figure 8 illustrates that non-virtualized environments tend to calculate more bogus operations than virtualized environments under similar stress conditions. This difference suggests that virtualized environments provide less processing performance to the stressors, which results in fewer bogus operations to induce a comparable level of system stress. Notably, in the GNU/Linux+priority scenario, the higher priority assigned to the real-time application also limits resources available for other operations, resulting in a 4.02% decrease in the cumulative bogus operations compared to the GNU/Linux scenario.

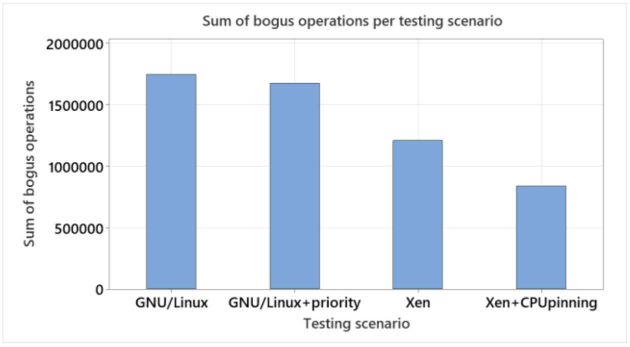

FIGURE 8. Sum of bogus operations per testing scenario.

The Xen and Xen+CPUpinning scenarios show the lowest total bogus operations compared to the non-virtualized scenarios. Specifically, the Xen scenario exhibits a 30.82% reduction in operations compared to the GNU/Linux scenario, while the Xen+CPUpinning scenario demonstrates a significant reduction of 51.97% in bogus operations compared to the GNU/Linux scenario. These reductions can be attributed to the resource management and allocation capabilities of the hypervisor in virtualized environments. Furthermore, the employment of CPU pinning technique in the Xen+CPUpinning scenario leads to a 30.57% reduction in bogus operations compared to the Xen scenario, due to the dedicated CPU resources allocated to each virtual machine.

7. CONCLUSIONS

This paper investigated the application of a virtualization technology on a processor intended for small satellites. The performed study involved conducting various experiments on the ZCU104 evaluation board, which is equipped with the same MPSoC as the Fraunhofer EMI's DPU, a high-performance data processor for small satellite missions.

Overall, the findings of this paper highlight the advantages and disadvantages of using virtualization. Virtualization with CPU pinning can significantly improve the predictability of a system under stress, delivering consistent execution times for real-time applications with strict timing requirements. This is shown in the Xen+CPUpinning environment which provided the smallest execution time's mean and variance when the application is executed in a separated, virtualized environment with CPU pinning. Furthermore, among the scenarios tested, Xen+CPUpinning showed the least variation in used memory. However, this scenario introduced additional overload in the system.

This study showed the advantages of virtualization in ensuring isolation between real-time applications and complex data processing tasks running in different virtual machines. The results illustrate the potential of virtualization techniques to improve the software reliability and performance of small satellite payloads based on a MPSoC. Virtualization can allow strict separation between different parts of the software which can be used in onboard applications to encapsulate mission-critical and non-critical software modules in virtual machines to ensure isolation. Furthermore, virtualization allows to run applications with different reliability levels simultaneously on the same device. In a small satellite, this can enable different users to access and use the same hardware platform while maintaining isolation between the small satellite's flight software and the different VMs where users can run their applications. With this approach, virtualization can be utilized as a possible satellite-as-a-service method, where small satellite operators can provide a shared infrastructure to enable sharing of the costs associated with developing or launching a satellite. In conclusion, this is a promising NewSpace strategy, where virtualization can contribute to lowering the barriers for newcomers and small satellite operators. This shared infrastructure fosters collaboration between different agents and space institutions, who will be able to easily join forces and work together on a mission.

8. REFERENCES

[1] D. Franzim, M. Ferreira, and F. Kucinskis, "A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions," *Journal of Aerospace Technology and Management 11*, doi: 10.5028/jatm.v11.1081.

[2] M. Grubb, "Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System," West Virginia University, 2021. [Online]. Available: https://researchrepository.wvu.edu/cgi/viewcontent.cgi?article=9091&context=etd

[3] K. Schäfer, C. Horch, S. Busch, and F. Schäfer, "A Heterogenous, reliable onboard processing system for small satellites," in *2021 IEEE International Symposium on Systems Engineering (ISSE)*, Vienna, Austria, 2021, pp. 1–3. [Online]. Available: https://ieeexplore.ieee.org/document/9582474

[4] M. Masmano, I. Ripoll, and A. Crespo, "XtratuM: a Hypervisor for Safety Critical Embedded Systems," in *11th Real-Time Linux Workshop*.

[5] A. Crespo, M. Masmano, J. Coronel, S. Peiró, P. Balbastre, and J. Simó, "Multicore partitioned systems based on hypervisor," in *19th World Congress The International Federation of Automatic Control*, 2014, doi: 10.3182/20140824-6-ZA-

1003.02410.

[6] *11th Real-Time Linux Workshop*. Dresden, Germany.

[7] S. Vanderleest, “ARINC 653 hypervisor,” in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, doi: 10.1109/DASC.2010.5655298.

[8] S. Vanderleest and D. White, “MPSoC hypervisor: The safe & secure future of avionics,” in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, doi: 10.1109/DASC.2015.7311448.

[9] S. Pinto, A. Tavares, and S. Montenegro, “Space and time partitioning with hardware support for space applications,” in *Data Systems in Aerospace Conference, DASIA 2016*.

[10] H. Karacali, N. Dönüm, and E. Cebel, “Xen Hypervisor Network Management System,” *The European Journal of Research and Development*, early access. doi: 10.56038/ejrnd.v3i1.244.

[11] Linux. “stress command.” https://linux.die.net/man/1/stress (accessed Jun. 23, 2023).

[12] Learn Linux Organization. “Scheduling, Priority Calculation and the nice value.: Chapter 7. System Tuning.” https://www.learnlinux.org.za/courses/build/internals/ch07s02.html (accessed Jun. 23, 2023).

[13] Alibaba Cloud Bao. “What Is Cpu Pinning In Virtualization.” https://www.alibabacloud.com/tech-news/virtualization/3ai-what-is-cpu-pinning-in-virtualization (accessed Jun. 26, 2023).

[14] Ninad. “What is Load Average in Linux?” https://www.digitalocean.com/community/tutorials/load-average-in-linux (accessed Jun. 26, 2023).

[15] Ubuntu. “stress-ng.” https://wiki.ubuntu.com/Kernel/Reference/stress-ng (accessed Jun. 26, 2023).