Rapture

An Efficient Cloud Gaming Platform Built on Containerization

E.F.J. Russel

Technische Universiteit Delft

TU Delft

Rapture

An Efficient Cloud Gaming Platform Built on Containerization

by

E.F.J. Russel

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 8, 2022 at 10:00 AM.

Student number:	4973976	
Project duration:	Aug 27, 2021 – July 8, 2022	
Thesis committee:	dr. J. Rellermeyer,	TU Delft, supervisor
	Prof. dr. ir. D.H.J. Epema,	TU Delft
	Dr. A. Katsifodimos,	TU Delft

This thesis is confidential and cannot be made public until

TU Delft

Abstract

Cloud gaming is a new paradigm that allows users to play games in the cloud and stream them to a thin client. While there is little research about cloud gaming, containerization technologies such as Docker could provide a virtualization alternative to Virtual Machines, as these suffer from overhead and GPU pass-through constraints. With Rapture, we provide an efficient cloud gaming platform based on containerization. Stable GPU multi-tenancy is achieved by resource restrictions such as frame rate and resolution limiting. A resource-aware best-fit scheduling algorithm accomplishes workload placement in the cloud gaming cluster. The best-fit algorithm outperforms other scheduling algorithms in node utilization and preservation of Quality of Experience. Furthermore, Checkpoint and Restore technologies enable migration, maintaining a high node utilization when down-scaling services. While migration improves efficiency, Quality of Experience is affected.

Preface

The past few months have been the final and though journey arriving at the end of my academic career. I began my studies in Mechanical Engineering, only to later find out my heart lay closer to Software Engineering. I have had eight years of studying behind me at the point of delivering my thesis. Writing and researching this thesis has been challenging, I do not consider myself an academic but more of an unhinged projectile when it comes to getting the job done. In the later stages of my thesis, I had to keep discipline and cancel social events and side projects to become more comfortable in writing, researching, and seeing the finish line more clearly. In the end, I am proud of this journey and, therefore, proud of the end result. In the simple but powerful words of the Vietnamese Thiền Buddhist monk Thích Nhất Hạnh, who passed away during the course of this thesis: *"This is it."*

I want to thank my supervising professor, Jan Rellermeyer, as he agreed to guide me in the process of experimenting with my own interests. Even though you transferred to another University in Germany, there have only been a few occasions where you were not able to meet me on a weekly basis. My gratitude goes to you for not having restricted me too much in my process.

Then, I want to thank my family and especially my parents. You have shown me, with all your love and support in any way, shape or form, how to become the person I am today. I am eternally grateful to be your son, and I hope I make you proud every day. Next are my friends: some I have known for years, some I only got to know when I started my bridging program at the TU Delft. You inspire me every day and are the creators of the perceived world around me. While I have been more and more confined to my desk towards the end of this thesis (and pandemic), I cherish each of your friendships as a piece of myself.

Finally, Linda. Words cannot express my thankfulness for you being in my life; I could not have done this without your love, support, comfort, and presence. You are the sunshine that makes my day.

E.F.J. Russel
Scheveningen, June 2022

Contents

1 Introduction

Video games have been increasing in popularity for over fifty years. Since the 1970s, when popular arcade games like Pong and the first video game consoles became available to consumers, the industry has matured into a multi-billion dollar market.

Nowadays, for a consumer to advance into video gaming and play the latest game titles, one needs a PC that should have adequate technical specifications for running a video game, or a video game console such as the Playstation, XBOX, or Nintendo. These video game consoles often feature high-performance components which come at a price. Moreover, during the current chip shortage, these consoles and components, such as Graphical Processing Units (GPU), are scarcely available.

Cloud Gaming is the practice of offloading the game to the cloud and letting the user play the game via a thin streaming client. The user will benefit from requiring a less powerful personal computer, a tablet computer, or even a smartphone. Thus, it omits the upfront purchasing cost of dedicated gaming hardware. Moreover, it could even discard the cost of purchasing a video game by allowing renting the video game via a subscription-based service. Another advantage of cloud gaming is the instant playing feature of cloud gaming. A user can purchase a game online and does not have to download it to their system first, which can be a lengthy process given the size of some games.

The benefits for game developers are also considerable. Due to offloading the game to the cloud, support for client platforms and devices incurs fewer development costs as they only have to run a streaming client. There is more control over hardware/software integration incompatibility issues; development only has to be done for a handful of cloud machines, reducing production costs. Furthermore, it aims to increase net revenues as there is no piracy when the executing environment is in the developers' hands. Finally, pushing updates and new content to existing games is much easier, as the developer has complete control over which version it presents to the user.

Current cloud gaming systems include Google Stadia, NVIDIA GeForce Now, and XBOX Cloud Gaming, with numerous market reports forecasting significant growth in cloud gaming services in the coming decade[1][2]. However, there is not much known about the operation and management of these specific systems.

Cloud compute providers such as Google Cloud, Amazon AWS, and Microsoft Azure have a global infrastructure for on-demand compute on a pay-as-you-go basis. One can rent a cluster of cloud instances, and these machines can be configured with GPUs for workloads such as training machine learning models or other high-performance computing (HPC) tasks. These dedicated GPUs are often capable of performance far exceeding what is necessary for a video game. Therefore, a single machine could host multiple video game workloads in a cloud computing setting.

Virtualization technologies, such as containerization, allow application packaging and execution in a loosely isolated environment, sharing kernel resources with the host machine. Containerization allows for easy deployment on cloud infrastructures, and a single machine can run several containers simultaneously due to low overhead. Containers can be configured to leverage resources from the

[1] https://www.gminsights.com/industry-analysis/cloud-gaming-market
[2] https://www.psmarketresearch.com/market-analysis/cloud-gaming-market

host, such as networking or hardware acceleration with GPUs.

In this work, we present Rapture, named after the eschatological belief that one will ascend into heaven, much like our trusted video game consoles and personal gaming computers will eventually reside in the cloud. Its name also corresponds to the underwater city in BioShock, one of the games I immensely enjoyed as a teenager.

Rapture is a cloud gaming platform that enables running Windows and Linux video games. It leverages containerization with Docker and other virtualization techniques to create a stable multi-tenant environment on a single hardware-accelerated cloud computing device. Furthermore, it employs a scheduling strategy on Docker swarm mode that ensures tight packing on these respective machines, resulting in high utilization rates per active computing device. Moreover, it incorporates live migration mechanisms to maintain a tight-packed system, even when downscaling services.

This research proposes a system that leverages existing cloud computing infrastructures for cloud gaming. We investigate components of this system and how they achieve isolation and efficiency. We compare the efficiency of the resource-aware scheduling strategy of Rapture to well-known container scheduling strategies. Finally, we propose a novel solution using inter-process communication (IPC), enabling Checkpoint and Restore for hardware-accelerated graphical applications to facilitate live migration of workloads.

1.1. Problem Statement

We identify three key points to building an efficient cloud gaming system with containerization technologies.

- **Isolation.** Virtual machines benefit from high isolation between instances on a single machine. With GPU passthrough and GPU hardware virtualization technologies such as NVIDIA GRID or AMD MxGPU; this isolation extends to hardware-accelerated resources as well. However, The aforementioned technologies divide the GPU into even numbers of identically sized profiles, and services like NVIDIA GRID also incur licensing fees adding to the cost. Video games feature a vast difference in required specifications for stable running on a system. Software virtualization, such as containerization and resource restriction, allows for tailored solutions and generally less overhead. Nonetheless, containerization has loose isolation, and impose security problems that need to be considered. A user should not experience a drop in FPS or an unstable gaming experience when other users occupy or vacate the shared host.

- **Scheduling.** Container orchestration frameworks such as Docker Swarm and Kubernetes incorporate general scheduling strategies. These strategies can be extended with resource reservations, such as required CPU and Memory, which affect the workload placement. These schedulers do not support GPU resource reservations out of the box, which is vital to prohibit overscheduling on a single hardware-accelerated machine. We also seek a scheduling strategy that accounts for tight-packing and high utilization per machine instead of spreading the workload for redundancy as in general scheduling strategies. We deal with heterogeneous and stateful workloads and not stateless replicated workloads that benefit from resiliency.

- **Migration.** Docker incorporates transparent Checkpoint and Restore technology with Checkpoint and Restore In Userspace (CRIU). Live migration for stateful container workloads that are non-reliant on external resources has been solved by leveraging CRIU and orchestration frameworks. There is little research about transparent Checkpoint and Restore for hardware-accelerated graphical applications. Moreover, this has not extended to containerized applications and live migration. After scaling a cluster to account for rising demand, decreasing demand will lower utilization per machine. Game containers of leaving users are stopped, and a residue of remaining running containers will be scattered over the over-provisioned cluster. One can take on over-provisioning a cluster by migrating game containers. Underutilized machines can be evacuated with migration and shut down, reducing costs. The evacuated workload is divided over the rest of machine nodes, ultimately regaining a tight-packed and properly provisioned smaller cluster. Furthermore, in mobile gaming migration is essential when the user is moving between edge-locations. In this manner, latency can be reduced by migrating the user to the closest edge-location.

1.2. Research Questions

Considering these points mentioned above, we define the following research questions, which we will address in this thesis.

RQ1 What virtualization techniques can we implement to support multi-tenancy in a cloud gaming system?

RQ2 Can we design a scheduler for cloud gaming systems?

RQ3 Can we enable Checkpoint and Restore for Hardware-accelerated 3D Graphics?

RQ4 Can we facilitate efficient down-scaling by migration in a Cloud Gaming System?

1.3. Organization

In this thesis, we will answer these research questions as follows: For **RQ1** we will give background into virtualization techniques in Section 2.2 and propose the techniques used in Rapture in Section 3.2. These techniques are analyzed in the experiments of Section 4.1. **RQ2** is answered by taking our containerized gaming instance to a cluster of machines. In Section 3.3 it is elaborated how these video game instances can be efficiently scheduled and managed while demand is varying. This section also incorporates migration strategy, which accounts for **RQ4**. However, an important element of enabling migration emerges from answering **RQ3**. A novel implementation that enables Checkpoint and Restore for hardware-accelerated 3D graphics is taken from inspiration out of Section 2.5 and thoroughly elaborated in Section 3.4.

2

Background and Related Work

In this chapter, we first elaborate on the key themes that are taken into account designing Rapture. It starts with history and background of video gaming in Section 2.1 and transitions to cloud gaming with the information in Section 2.2 and Section 2.3. A more in-depth overview is made of state-of-the art Checkpoint and Restore technologies in 2.4 which is further discussed in 2.5, as well as other works that influence this thesis.

2.1. Video Games

A video game is a game played by electronically manipulating images produced by a computer program on a display. The history of video games dates back to as early as 1947, when simple games were developed with cathode-ray tubes as the display. The first multiplayer video game, Tennis For Two (1958), was an analog computer linked to an oscilloscope played by two people using hand controls. Tennis for Two predates the popular arcade game Pong (1972), which later became available as a game console for home tv sets (1975)[1]. The first true 3D game was considered to be Battle Zone (1980), a first-person tank shooter. It used three-dimensional wireframe vector graphics displayed on a black and white vector monitor.

The development of ”online” multiplayer games began with Maze War (1972), with the first versions featuring a peer-to-peer protocol via a serial connection, and later versions from Xerox featured Xerox Network Systems (XNS) over Ethernet. However, SGI Dogfight (1985), an aviation-inspired game for Silicon Graphics workstation computers, can be considered the first true online multiplayer game as we know them today. It had support for UDP, making it the first game to use the Internet Protocol Suite, and later IP Multicast capability was added to be played across the internet.

Figure 2.1: An advertisement for Super Pong IV, a four player variant of the popular Pong game.

[1] http://www.pong-story.com/intro.htm

The first video games were dedicated machines with a single preprogrammed game; this has evolved into many reprogrammable platforms, including gaming consoles such as the XBOX and PlayStation, handheld gaming consoles such as the Nintendo GameBoy, the personal computer, and mobile devices. Reprogrammable video game systems first used cassette tapes, followed by dedicated ROM cartridges and memory cards. More recent media, such as optical discs, are also being replaced by digital distribution with downloading and streaming games.

Interaction with a video game is often done via peripheral devices such as gamepads and joysticks, mice and keyboards, and touch screens. Also, dedicated controllers such as steering wheels, aviation yokes and cameras such as the PlayStation Eye Toy and XBOX Kinect are not uncommon. The display is either built-in to the gaming device, or an external TV screen or computer monitor is necessary for connecting the gaming platform. Virtual Reality (VR) gaming has also been rising in popularity. In VR, the game is displayed on a pair of 3D goggles for complete immersion into the game.

A central part of live-action video games is the game loop. While typical computer programs only respond to user input and instructions, a video game requires the user to participate in the game environment, which often continues regardless of user interaction. In the game loop, the game checks for user input events and processes network updates of both the user and opponents in an online multiplayer game. Furthermore, it calculates the new positions of the camera and objects in the scene and resolves environment actions, such as collisions and artificial behavior. Finally, it plays sounds and draws graphics to the screen.

Operations of the game are split between the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). The CPU is responsible for handling most of the game's logic and sending drawing instructions to the GPU, which uses this information to render the image on the screen.

2.1.1. Graphics Processing Unit

Hardware acceleration refers to offloading computation to a specialized electronic circuit or hardware component (graphics card). A GPU is a specialist processor designed to accelerate the creation of images in a frame buffer, which is outputted on the display.

The graphical workload that the GPU accelerates consists of many floating-point operations and Matrix arithmetic. These individual calculations generally are not interdependent and can thus be processed in parallel. The GPU processes large blocks of data in parallel more efficiently than a CPU, which in turn is superior in sequential operations.

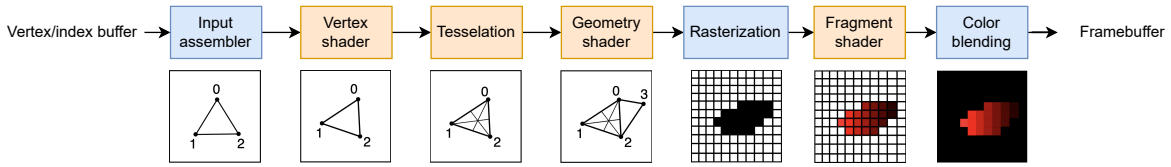


Figure 2.2: Graphics pipeline of Vulkan, a cross-platform Graphics API

In video gaming, the GPU handles the rendering or graphics pipeline. This pipeline defines what steps a graphics system needs to perform to render a 3D game scene to a 2D screen. It often is a two-stage process; in the first stage, the graphical models and textures need to be loaded into the memory of the GPU called Video Random Access Memory (VRAM). This is necessary as the GPU cannot directly access the computer's Random Access Memory (RAM). In the second stage, positions and other information about objects and in-game cameras are used to draw the image.

More specifically, this second stage takes the graphical models, consisting of a triangular mesh, and applies coloring in the form of textures and materials. Furthermore, it draws lighting into the scene, which can achieve a photo-realistic effect through techniques like ray tracing [1]. Finally, it determines the position of the camera in the scene. These make up the field of view of the output image and determine the color of the display pixels. The graphics pipeline is an abstraction of the actual operation steps on the GPU; these instructions are usually given to the GPU via a higher level graphics library or API.

2.1.2. Graphics Libraries & APIs

Developing computer games can be done in any programming language; nevertheless, when one wants to develop video games with 3D graphics and use hardware acceleration, there is a need to communicate conveniently with the GPU for drawing images. This is where Graphics Libraries and Application Programming Interfaces (APIs) come into effect. The graphics pipeline operations, elaborated in the previous section, depend on the underlying software and hardware. Graphics APIs provide an abstraction of the underlying hardware and unify the steps to control the graphics pipeline of a given hardware accelerator.

There are diverse Graphics APIs with applicability to different platforms and operating systems. Most notable are:

- **OpenGL**[2] OpenGL, developed by Khronos Group, is the most widely adopted 2D and 3D graphics API. OpenGL is language-independent and cross-platform, with some noteworthy bindings being WebGL (Javascript), WGL, GLX, and CGL (C) and the bindings for mobile platforms such as iOS and Android. The OpenGL context is obtained and managed by the underlying window system. Thus, there are no APIs provided for windowing, but also not for audio or input. While the programmer can implement the API entirely in software, it is designed to be implemented mostly or entirely in hardware.

- **Direct3D** Direct3D is a graphics API part of DirectX, a collection of multimedia APIs of Microsoft Windows. Direct3D11 and Direct3D12 are current versions widely used in video game development. Since most video games are developed for Windows, Direct3D can be considered the most popular graphics API in gaming.

- **Vulkan**[3] Initially developed by AMD as Mantle, but further developed as Vulkan by Khronos Group as a follow-up of OpenGL. Vulkan strives to offer lower overhead and more direct control over the GPU; it can be categorized as a lower-level Graphics API because more computing is offloaded to the GPU. The CPU benefits from lower usage thanks to reduced driver overhead. Like OpenGL, Vulkan is cross-platform, but there is a unified API, whereas, with OpenGL, these were split for both desktop and mobile graphics devices. Furthermore, Vulkan is multithread-friendly, which helps with scaling on multi-core CPUs. It uses precompiled shaders, which OpenGL uses a high-level approach that needs a separate compiler. Shader pre-compilation speeds up application initialization speed and the number of distinctive shaders used per scene. Finally, the unified management of compute kernels and graphical shaders eliminate the need to use a separate computing API in conjunction with the graphics API.

In this research, we primarely focus on OpenGL and Vulkan. As OpenGL provides a simple entry to 3D graphics, Vulkan shows excellent cross-platform performance.

2.1.3. Cloud Gaming

Modern video games strive to be as photorealistic and exciting as possible. This trend naturally requires the gaming platform to feature the latest specifications, adhering to this experience. The upfront purchasing costs of such a gaming platform, whether it is a PC that should have adequate technical specifications or a video game console, are often high.

Cloud Gaming is the practice of offloading a video game to a server in a data center. The server executes the video game headlessly. Either video images or graphics instructions [2][3][4] are sent over a high-bandwidth and low-latency network via a media stream. From the user, game input controls are received via the web. This allows the user to play the game via a thin client. By thin, it is meant that the device does not have considerable computing and graphics capabilities. Thus, such a client could be a less powerful personal computer, a tablet computer, a smart TV, or even a smartphone. Nevertheless, streaming over the network requires high bandwidth, induces latency, and sometimes packet loss. Latency and bandwidth limitations are the primary restraining factors in cloud gaming. However, the development of fiber-optic internet infrastructures and 5G cellular bandwidths pose a bright future.

[2] www.opengl.org
[3] www.vulkan.org

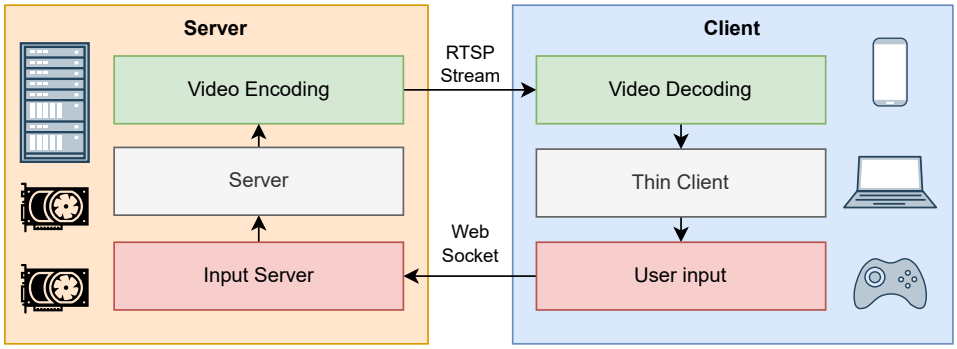


Figure 2.3: Schematic of a Cloud Gaming system

Cloud gaming benefits users by enabling gameplay on previously not capable devices. It also introduces new possibilities such as instant playing for the user without the need to download and install after purchasing the game. From the game developer's perspective, there is more control over the execution environment and less compatibility development needed for different hardware. It also eliminates piracy, and the developer has complete control over updates as it has full control over which version of the game it presents to the user.

Commercial game streaming clients that a consumer can use to run video games on their gaming PC and stream them to a TV or handheld device is Parsec[4] and Steam Remote Play[5]. But these still require the ownership of gaming hardware.

On-demand or subscription-based cloud gaming platforms include Google Stadia, NVIDIA GeForce Now, and XBOX Cloud Gaming. These platforms offer game selections at different subscription prices. For these services, only a thin client with a video game controller is needed. The video game console is virtualized and abstracted away from the user.

2.2. Virtualization

Servers and computing devices in data centers are often powerful and can handle many tasks at the same time. It is, therefore, common practice to provide multiple users a piece of computing, storage, or network resources of a single machine. This is referred to as Multi-tenancy.

Virtualization is the act of creating an abstract layer and dividing physical hardware resources, such as CPU, Memory, and Disk, into logical parts. In a multi-tenancy situation, each user is assigned one or more of these logical parts that behave as if they were physical resources. Virtualization results in a flexible, scalable, and tailored solution for the demand. It increases efficiency and lowers the cost per user.

It is, however, essential to maintain a high level of isolation and security; the users may never notice each other while they are operating on the same machine. Consider a demanding workload for one user that overloads the system and leaves no resources for the other users. This preservation can be enforced by software that carefully partitions and limits resources. But hardware-assisted virtualization is also commonplace, which uses resources such as the CPU or dedicated hardware directly to handle the workload of virtualization management software.

[4] parsec.app
[5] store.steampowered.com/remoteplay

2.2.1. Virtual machines and containers

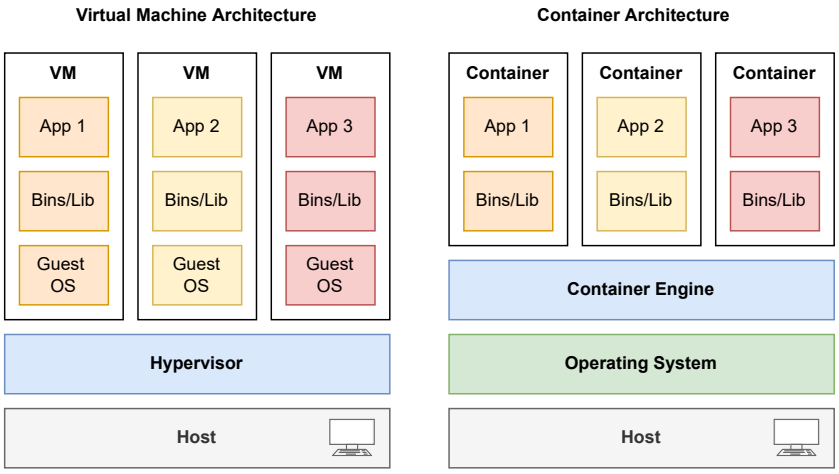


Figure 2.4: Virtual Machines versus Container Architecture

Virtual Machines
One can distinguish between hardware-level virtualization and operating system-level (OS) virtualization. Virtual Machines (VM) are a form of hardware-level virtualization which run on an abstraction layer called a virtual machine manager (VMM) or hypervisor. Every virtual machine runs its own OS and is logically isolated from other VMs on the system. The hypervisor virtualizes the underlying hardware and makes these resources available to the virtual machine. Hypervisors can be classified into bare-metal hypervisors, which operate directly on top of the hardware, and hosted hypervisors, which operate on top of a host's OS. Common hypervisors are VMware ESXi/vSphere, Microsoft Hyper-V, and Citrix XenServer.

Hardware-level virtualization can further be categorized into full virtualization, which virtualizes the actual hardware entirely and allows complete guest operating systems and apps to run unmodified. Paravirtualization does not simulate a hardware environment, and the guest operating systems need to be modified to work.

Containers
Containers are a form of operating-system-level virtualization. With OS-level virtualization, software applications run in isolated user spaces. These isolated environments are bundled with configuration files, libraries, and dependencies. The guest OS kernel is shared between these containers, so a hypervisor is not needed. However, this imposes more loose isolation than virtual machines and induces privacy and security concerns.

Control groups (cgroups) is a Linux kernel feature that allows for allocating, limiting and isolating resources. These resources include CPU time, system memory, network bandwidth, and disk usage. Hardware resources can be divided over tasks that run in these cgroups. Other kernel features, such as namespaces, provide system restrictions, and SELinux provides additional security mechanisms. Examples of OS virtualization-based systems include LXC, Docker, and Solaris Containers.

Performance
While the performance of hypervisors, such as KVM, has been increasing over the years, VMs still introduce disk input/output (I/O) overhead and lower CPU performance compared to containers. The overhead introduced by containers is almost negligible on CPU and memory usage and shows only a slight overhead on I/O and OS interaction [5][6] [7]. Furthermore, VMs run their own guest OS and thus suffer from slower boot-up times and require more disk capacity. Containers bring convenience, faster deployment, and elasticity.

2.2.2. GPU virtualization
Most virtualization solutions solve the sharing of hardware such as CPU and memory, but given the complexity of the GPU, these solutions are nontrivial and induce trade-offs. The reason is that GPU

specifications and drivers are highly secretive, and the generational cycle is short, with architectures constantly changing entirely. Therefore, a virtual device corresponding to a multitude of GPUs is a devious task. It is also the case that all graphics applications interact via standardized (Graphics) APIs that abstract away from the underlying hardware. While AMD does have available documentation for a decade now, NVIDIA drivers have been closed-source until recently. NVIDIA has released its first version of open source modules just now[6].

We can divide GPU virtualization into backend and frontend virtualization. Backend techniques have the virtualization boundary between the stack and physical GPU hardware. Frontend virtualization introduces a virtualization boundary at a higher level in the stack and runs the graphics driver in the host/hypervisor [8].

Backend virtualization

GPU pass-through is when the host presents an available GPU to the virtual machine. If the GPU is accessed only by a single virtual machine exclusively, we speak of Fixed Pass-through. If the GPU is divided into a set of dedicated smaller contexts or partitions, it is named Mediated Pass-through.

- **Fixed Pass-Through.** Provides exclusive access to the virtual machine and can achieve high performance levels close to native configurations. In the case of a host running multiple VMs, the GPU is only assigned to one VM and cannot be shared, which means every VM needs a dedicated GPU, which is a costly option.
- **Mediated Pass-Through.** Dedicates a set of contexts for VMs to use on the host. A GPU can be partitioned into smaller independent contexts; each can be assigned to a VM by the hypervisor. This is also a form of hardware-assisted virtualization. Examples of this are NVIDIA GRID, which exposes vGPU profiles to be assigned to VMs. The technology incorporates a memory management unit that keeps the memory profiles separate and dedicates independent input buffers to different VMs, separating each VM's command stream into independent rendering contexts [9]. AMD has a similar technology called MxGPU, which uses Single-Root I/O Virtualization (SR-IOV) to separate access to resources among various PCIe hardware functions [10].

Performance of GPU passthrough technologies requires optimization to achieve near-native performance with different types of workloads and has been improving in the past years. The performance also differs between hypervisors and GPUs [11][12]. While general purpose GPU (GPGPU) workloads leveraging CUDA and OpenCL mostly show high-performance numbers, gaming benchmarks indicate a memory bottleneck which is accounted for by the constant data transfer between GPU and VM. Commands that operate exclusively on the GPU are much less susceptible to overhead created by the virtualization system; this is apparent looking at the ray-tracing performance, which achieves closer to native results [13]. NVIDIA GRID graphics cards are optimized for cloud gaming and should achieve closer to native passthrough performance in gaming workloads [14].

Finally, mediated passthrough technologies are not available to consumer-grade graphics cards, and the configurations divide the GPU into even numbers of identically sized profiles, and services like NVIDIA GRID also incur licensing fees adding to the cost.

Frontend virtualization

Frontend virtualization can be divided into API-remoting, which forwards graphics API calls to an external graphics stack, and emulation, which synthesizes host graphics operations in response to actions by the guest device drivers.

API-remoting introduces overhead due to communication and serialization to cross the virtualization layer to the host, which executes the calls via graphics APIs to the GPU. But in the case of containerization, as the host OS kernel is shared, The application can access the graphics API directly from within the container.

With the release of Docker 19.03, Docker runtime natively supports NVIDIA GPUs and allows the user to run applications in a hardware-accelerated container. The performance of hardware-accelerated applications in containers shows near-native performance, including video games [11][12][15]. Since containerized video games utilize the host machine's resources more efficiently than a virtual machine running the video game, it can allow for more game instances per host device and a better Quality of Experience (QoE). This leads to cost savings, but again, security and isolation concerns remain.

[6] https://developer.nvidia.com/blog/nvidia-releases-open-source-gpu-kernel-modules/

Resource restrictions
The mentioned cgroups limit a container's CPU time, system memory, network bandwidth, and disk usage. Likewise, one could impose resource restrictions per container sharing a single GPU in the form of space-sharing and time-sharing.

Space-sharing limits the allocation of VRAM by applications to the container or VM to reserve resources for that particular application [16] [17]. Time-sharing allocates time per application to access the GPU and request resources; a container can get complete GPU performance for a period [18]

Most of these solutions rely on wrapping the graphics and computing API calls sent to GPU and monitoring and altering the resource allocations made by the application. Sharing a GPU in containers shows better performance in video gaming than Mediated Pass-through solutions [19] and is more flexible since resources can be allocated in a fine-grained manner. Finally, it is available for any consumer graphics card and is not susceptible to licensing fees.

In gaming, one could lower the required VRAM of a video game by reducing the resolution as the frame buffer is stored in VRAM. Lowering rendering quality settings also helps as the model scene, which includes textures and geometries, is loaded into VRAM. Lower quality versions of these assets take up less space[7]. Limiting resolution and quality per video game allow for more video games to be placed in VRAM.

Framerate limiters for video games halt drawing to the frame buffer for a short period not to exceed the limited frames per second. Since these operations require GPU bandwidth, lowering the frames per second reduces this bandwidth, and multiple games could time-share bandwidth by restricting frames per instance.

2.2.3. Compatibility layers
While containers show interesting capabilities for running multiple video games on a single host, container technologies are primarily available under Linux. There is a windows version of Docker, but it essentially runs under a Linux subsystem or cannot access GPU capabilities.

Compatibility layers allow applications requiring a specific OS to be run on a different OS by translating system calls. Wine [20] is a prevalent compatibility layer and allows users to run programs compiled for Windows under Linux, MacOS, FreeBSD, and NetBSD. Wine is a single process that translates Windows system calls to Linux system calls. It is also responsible for adequately loading Windows applications. Wine supports 64-bit, but also 32-bit and 16-bit legacy applications

Extensions to Wine allow support for dedicated APIs such as DXVK, which translates Direct3D calls from Windows applications to Vulkan API calls[8]. Proton[9] is based on Wine and DXVK and allows users to play Windows games on Linux with Steam, with ProtonDB[10] a database where users can give a rating about the performance of specific video games under Proton. Running video games under Wine and DXVK allows Windows games to be executed in Docker containers on a Linux system alongside native Linux games.

2.3. Cloud computing
Cloud computing enables the offloading of computing resources to a service provider. Cloud computing aims to offer consumers and businesses on-demand storage and computing on a pay-as-you-go basis. It relieves the burden of server and database management and often provides a wide range of scaleable technologies useful for businesses and consumers. Cloud computing uses many virtualization techniques to offer a fine-grained custom solution for customers.

Large cloud service providers like Google Cloud, Amazon AWS, and Microsoft Azure have a global infrastructure with resilient geolocations worldwide. These geolocations are often large data centers with many machines for a single region in a country or continent. Another cloud computing paradigm is Fog or Edge computing, which strives to put computing as close to the user as possible; this technique offers low-latency connections to smaller server farms and a favorable model for Cloud Gaming and real-time streaming applications.

Cloud computing primarily falls into the following three layers.

[7] www.cgdirector.com/how-much-vram-do-you-need
[8] github.com/doitsujin/dxvk
[9] github.com/ValveSoftware/Proton
[10] www.protondb.com/

- **Software as a service (SaaS).** SaaS usually comes in the form of cloud applications accessed through the browser, which we use in our daily lives, such as e-mail, video calling, and text editors. This is an alternative to client applications that run on a personal computer. The service provider has total control over these SaaS applications and offers accessibility from multiple devices, including smartphones, and cloud backup.
- **Platform as a service (PaaS).** The service provider often has modular cloud applications such as network gateways, databases, access and identity management, and monitoring services with PaaS. The service provider manages all of the security and infrastructure for these services. This allows consumers to shift focus on building the core logic of their business.
- **Infrastructure as a service (IaaS).** IaaS gives the user the highest level of control, in which he or she can demand (virtualized) infrastructure in the form of computing, network, and storage, among others. This allows direct access to the cloud machines in the data center and requires its setup and maintenance to be done manually.

2.3.1. Container orchestration frameworks

In an IaaS setting, operating a cloud infrastructure, answering demand and scheduling workload for users requires setting up a system that divides this workload over the available resources in a data center. With orchestration tools, the cluster is configured, managed, and coordinated in an automated manner. For containerized applications, container orchestration tools are mainly used for scheduling and managing container lifecycle. Container orchestration tools are often used in micro-service architectures. Two widely used container orchestration frameworks are Docker Swarm and Kubernetes. When one has a set of (virtual) computing machines on a network, these technologies offer an easy setup of a cluster.

- Docker Swarm mode[11] (previously Docker Swarm and Swarm Kit) is included natively in recent versions of Docker. It offers scaling, desired state reconciliation, multi-host networking, service discovery, load balancing, and rolling updates. Compositions of containers can be deployed on the cluster as services or stacks, which are configured in .yaml files.
- Kubernetes[12] is an open-source container orchestration tool initially developed by Google. Kubernetes used Docker runtime for running containers but moved on to directly interface the container runtime via Containerd. It offers more capabilities than Docker Swarm, including automatic (horizontal) scaling, self-healing, multi-host networking, service discovery, load balancing and rollouts and rollbacks. It is also designed for extensibility. Containers are deployed in so-called pods, and the configuration is done in .yaml files.

Docker swarm is lightweight and easier to set up and control than Kubernetes; interfacing is through the Docker CLI, requiring no other CLI knowledge. However, it offers fewer automatic scaling and self-healing features than Kubernetes. Furthermore, Kubernetes is open-source, extensible, and cloud providers offer easy hosting.

2.3.2. Scheduling

The task of a scheduler in cloud computing is to optimize performance and reduce cost by properly placing workload regarding requirements and preferences. When launching a new container instance, the container orchestration framework must choose which node in the cluster will host this workload. Both Docker Swarm and Kubernetes automatically schedule the service or pod onto a node by a set scheduling strategy.

- Docker Swarm implements a spreading strategy, which strives to divide the workload over the cluster to maximize availability and resiliency. The user can specify constraints to which set of nodes with a specific label the container is scheduled to (for instance, only nodes that have SSDs as storage) and preferences that only come in the form of spreading over a set of nodes with a specific label. Multiple preferences determine the sorting order in spreading, for instance,

[11] docs.docker.com/engine/swarm
[12] kubernetes.io/

spreading first over geolocation, then server racks and servers within the rack. Resource reservations can also be made, which request available CPU and Memory to be used exclusively by the service.

- Kubernetes has a more complex scheduler that uses predicates for filtering and priorities for scoring. These determine the so-called node-affinity that the Kubernetes scheduler uses to define the placement logic of pods on specific nodes. Like Docker Swarm constraints, these predicates allow scheduling pods on nodes with specific capabilities. The scheduler extender can adapt the behavior of the Kubernetes scheduler, or because the source code is open source, one can also modify the scheduling implementation.

The spreading strategies of these container frameworks are beneficial for stateless and replicated workloads that benefit from resiliency and load-balancing, which is often the case in microservice architectures. In cloud gaming, we deal with stateful and non-replicated workloads; thus, an adapted strategy would be more fitting.

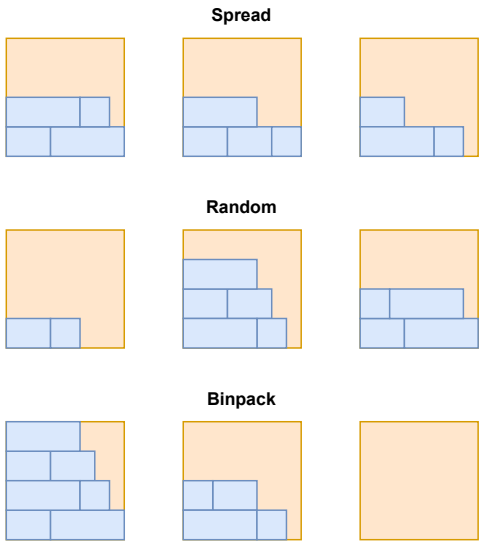


Figure 2.5: Spread, Random and Binpack strategies

Previous versions of Docker Swarm incorporated more widely known scheduling algorithms such as Binpack and Random next to Spread. A Binpack algorithm places the workload on the first available machine, which could be favorable if one wants to utilize as few machines as possible. In a cloud gaming environment where machines with a GPU are costly to operate, tight-packing and high utilization per machine could drive down cost [21].

Nevertheless, the legacy and newer scheduling algorithms do not support GPU scheduling out of the box, and partial GPU resource reservations are not supported; only a single GPU per pod or service.

2.3.3. Migration

In a cloud infrastructure sometimes it is needed to migrate a stateful process or workload from one machine to another. This could be the case when a host is not working properly and needs to shut down for maintenance, but the provided service still needs to be maintained. It could also be for less critical reasons such as rebalancing load. In cloud gaming and edge computing, migration would greatly benefit mobile gaming as a user could be on the move between different edge-locations[22].

Cold migration means that the VM or container is shut down before migration and restarted after migrating. Hot or live migration does not disrupt the service provided and remains available while migrating, and does not involve shutting down and restarting the VM or container. Seamless live migration can be enabled by streaming and synchronizing the necessary session data before migration. Live migration of VMs with GPUs is enabled for NVIDIA GPUs with Citrix Xenmotion or VMWare vMotion[13]. Container migration is not implemented in Docker, but Checkpoint and Restore technologies,

[13] https://www.nvidia.com/en-us/data-center/virtualization/virtual-gpu-migration/

and filesystem sharing allow for the migration of Docker containers [23].

2.4. Checkpoint and Restore Technologies

Checkpoint and Restore (C/R) technologies allow applications to make a checkpoint or snapshot during operation, from which the application can recontinue after shutting down, or a crash has occured in that application. This is necessary for failure tolerance in long-running applications, in which periodic checkpointing will ensure that if a crash occurs, the program can continue from the last checkpoint. Furthermore, it can enable the migration of applications as the application is checkpointed before migration and restored after the migration is complete. One can distinguish between application-level and transparent checkpointing solutions.

- **Application-level:** A simple save state or checkpointing in a video game is an example of application-level checkpointing. At a point in time, the user can manually save the game, or the game is saved periodically at which the next gaming session the user can resume from that checkpoint.
- **Transparent:** On the system level or in user space, transparent solutions can checkpoint the application without requiring source-level modifications or in-place application-level checkpointing solutions. Checkpoint and Restore technologies are said to be fully transparent if it requires no adaptation to the kernel or application for enabling Checkpoint and Restore.

Application-level checkpointing is more time and space-efficient than transparent C/R solutions as it can optimize the required data needed to restore the application state. However, it requires implementation and is not a universal solution. It will not work for legacy applications, which is necessary for a cloud gaming setting where one wants to support many already released titles. Different transparent C/R technologies have different implementation methods. These are BLCR, DMTCP, and CRIU, which we describe in-depth in sections 2.4.1, 2.4.2, and 2.4.3.

2.4.1. BLCR

Berkeley Lab Checkpoint/Restart [24] is a system-level C/R technology in the form of a loadable kernel module and a small library. It was developed to support distributed checkpointing for Message Passing Interface (MPI) in High-Performance Computing (HPC) environments. The implementation relies on the ”VMADump” command that maps shared virtual memory areas of source processes. These are dumped to a checkpoint file; it furthermore preloads a shared library into the application with a signal handler to manage callbacks for multiple parent and child processes of an application for checkpointing.

Because it runs on a system level, it can modify Process Identifiers (PIDs) upon restore to mitigate conflicts with currently running applications and has an excellent overview of open files of different applications. The kernel module is developed to be as lightweight and inependent as possible to allow maintenance with kernel changes easily.

2.4.2. DMTCP

Distributed MultiThreaded CheckPointing [25] is a user-level transparent C/R technology. Unlike BCLR, DMTCP does not require special kernel modules or kernel patches, which means it does not rely on controlled environments and kernel modules that have to keep up with kernel changes. DMTCP also requires no system privileges to operate, allowing it to be bundled with the application.

Like BCLR, DMTCP was developed for checkpointing distributed HPC workloads; it allows checkpointing of a network of processes spread over many nodes by first copying all of the inter-process information to user space, at which it consequently executes single-process checkpointing.

Programs are run under the `dmtcp_checkpoint` command, which registers it within the set of processes that need to be checkpointed. Checkpointing is added to the application by injecting a shared library that wraps around libc functions and system calls to record information about the creation of sockets and kernel state.

Checkpointing an application is done in the following four steps:

1. A checkpoint is requested from the user manager. User threads are suspended, and the owner of each file descriptor is saved. DMTCP waits until all applications in the network are suspended

2. Elect shard file descriptor leaders: DMTCP executes an election of a leader for each potentially shared file descriptor, with the last in line elected as owner.
3. Drain kernel buffers and perform handshakes with peers. The connection information table is then written to disk and DMTCP waits until all applications in the network are drained.
4. Write checkpoint to disk: The contents of all socket buffers are now in user space. DMTCP waits until all applications in the network are checkpointed.

Restoring an application takes the following seven steps:

1. Reopen files and recreate outputs: File descriptors, excluding sockets connected to a remote process, are regenerated first. These include files, listen sockets, uninitialized sockets, and pseudo-terminals.
2. Recreate and reconnect sockets: relocated processes are found with the discovery service.
3. Fork into the desired amount of user processes.
4. Rearrange file descriptors for user processes.
5. Restore memory and threads.
6. Refill kernel buffers.
7. Resume user threads.

DMTCP resolves PIDs of restored processes by a virtual PID table that allows restored dependent processes to function correctly. DMTCP also supports pipes and shared memory segments and can be extended with plugins that allow writing code functionality triggered by event hooks.

2.4.3. CRIU

Checkpoint/Restore In Userspace [26] does not require preloading its libraries like DMTCP, it also does not rely on intercepting and forwarding calls. The only requirement is a kernel providing the needed facilities, and CRIU features were added in recent kernel versions of Linux.

One can run CRIU with the `criu dump <pid>` command. **Checkpointing** is done as follows:

1. Collect the process tree and freeze it: The PID of a process group leader is obtained from the process tree. Using the parent PID, the dumper walks through the process task directory, collecting threads and gathering process children recursively. Tasks are stopped using `ptrace`.
2. Collect task resources and dump: All the available information about collected tasks is read and written to dump files. The resources are obtained via VMA areas are parsed, file descriptor numbers are read and core parameters of a task (such as registers and friends) are being dumped via `ptrace` interface and parsing status information of the process.
3. Injecting parasite code: CRIU injects a parasite code into a task via the `ptrace` interface. This is done in two steps: First, injecting only a few bytes for `mmap` system call at the CPU's location fetching task instructions. Then a system call is injected, and enough memory for a parasite code is allocated needed for checkpointing. The parasite code is then copied into a new place inside the to be checkpointed task address space, and execution is pointed to the parasite code. This parasite code allows CRIU to gather more information such as credentials and contents of memory.
4. After everything is dumped, such as memory pages, which can be written out only from inside the to be checkpointed task address space, clean up is done by using `ptrace` and removing all the parasite code, for the task to resume after checkpointing.

Restarting is done by CRIU morphing itself into the tasks it restores; the four basic steps are:

1. Resolving shared resources: CRIU reads the checkpoint image files to find out which processes share which resources. A single process restores shared resources and, in a later stage, reconnected.

2. Fork into the desired amount of user processes.

3. Restore basic tasks and their resources: CRIU restores all resources except for exact memory mapping locations, timers, credentials, and threads. Among others, CRIU opens files, prepares namespaces, maps, and private memory areas, and creates sockets.

4. Switch to restorer context, restore the rest and continue: Since CRIU morphs into the target process, it needs a small piece of code that removes all of CRIU's code and restore the application code. This code does not interfere with CRIU or restored application mappings. Finally, the timers, credentials, and threads are restored.

CRIU's restore operators do not require PID virtualization. Furthermore, CRIU allows for plugins and exposes endpoints to external UNIX sockets, external files, external bind mounts, and external net devices (links). Also, example plugins are provided for some external devices. Finally, CRIU is said to be experimentally supported by Docker since Docker 1.13, but tests show that later versions of Docker need the latest versions of Containerd and Runc built from source to operate on Linux.

These technologies incorporate different solutions for external devices and enable developers to extend these capabilities. Nevertheless, checkpointing GPUs is not trivial. It requires extensive knowledge of the hardware connected to the checkpoint application, such as virtual memory address (VMA) maps and checkpoint callbacks that need to be in place to allow the draining of operation queues.

2.5. Related Work

To build a state-of-the-art cloud gaming system, we resort to research and development conducted that will allow us to implement such a system. While there exists research about improving the streaming and bandwidth quality of cloud gaming systems [3][14], we are more focused on creating an efficient hosting platform.

Isolation and multi-tenancy

In the work of Chen et al. [19], the authors present TG-SHARE, an app container on Windows that allows for multiple games on a host with a single OS. TG-Share is built on Sandboxie, a free and open-source application isolation software for Windows that focuses on the security and privacy of untrustworthy applications. The work states that TG-SHARE uses ”OS functions” to monitor system resources, including GPU, and perform appropriate management. This research does not explain how this management is executed, and there are no Windows OS functions known for limiting GPU usage of applications.

A more straightforward example of monitoring GPU usage per container is portrayed in ConVGPU by Kang et al. [16]. ConVGPU actively wraps calls from the CUDA API that handle GPU resource allocation and deallocation. Combined with a GPU memory scheduler, the allocation call can either be approved or denied by the scheduler, which pauses execution if needed. If a running container requires GPU memory that is not available, every memory allocation requested by this container is suspended until the scheduler assigns more GPU memory to the container.

A similar piece of work named GaiaGPU, by Gu et al. [17] partitions a GPU into multiple virtual vGPU's and exposes them on the device plugin framework of Kubernetes. Each container can be assigned with one or more vGPUs as requested. Similar to ConVGPU, a scheduler and manager intercepts CUDA API calls. It imposes a hard limit (suspension) on memory allocation calls or an elastic limit that could utilize free resources in the GPU, currently reserved but not used by other containers. Experiments with multiple deep learning frameworks show low overhead with minor fluctuations in vGPU resources.

Kube-Knots by Thinakaran et al. [27] provides solutions within the Kubernetes framework, leveraging the device plugin. Knots, a GPU-aware orchestration layer, monitors GPU utilization metrics and allows for managing compute, which is time-shared while the memory is space-shared. By optimizing for dynamically harvesting spare compute cycles, the utilization per GPU in the cluster is kept at a high rate. The harvesting combined with scheduling techniques improves cluster-wide GPU utilization by up to 80 percent.

Scheduling

For scheduling, we focus on works that imply scheduling strategies that demonstrate high utilization per machine. In the work of Keni et al. [28], a cloud broker enabled by the ”Adaptive Containerization for Microservices in Distributed Cloud” (ACMDC) scheduling strategy strives to find the minimum amount of computing resources needed while serving requests from cloud consumers. The scheduler reacts to resource request events and reconfigures the existing deployment.

For a fixed set of 10,000 requests to deployed services, ACMDC shows better results than Binpack, using no more resources than needed unlike Spread and Random. The number of available machines does not dictate the number of machines in operation, as computing resources are limited to a small number of machines. This also shows a more linear relationship between the number of service requests and machines in operation.

Both the previously mentioned works ConVGPU [16] and Kube-Knots [27] employ a best-fit scheduling strategy when multiple containers request GPU memory resources. This strategy has shown better performance than strategies such as First-In-First-Out or Random, and significantly improves the average GPU utilization, letting other GPU nodes be in a deep sleep state. The Kube-Knots work also demonstrates that GPU energy consumption scales linearly with utilization; thus, it is natural to schedule for high utilization rates.

Checkpoint and Restore

As previously stated, enabling Checkpoint and Restore for applications interacting with GPUs is nontrivial. However, a working example is demonstrated in an AMD keynote by Kuehling et al. [29]. By using the plugin interface of CRIU they can synchronize the dumping of the GPU state with the dumping of the application state. This requires knowledge of the device VMA maps and endpoints in the GPU driver that allows draining of the state. Furthermore, this is only demonstrated for ROCm compute, which is similar to CUDA for NVIDIA and does not work for graphics APIs such as OpenGL and Vulkan.

The research group of Northeastern University that developed DMTCP posed solutions for numerous applications that leverage GPU capabilities. It started with external interfaces such as MPI in the work of Garg et al. [30] and later moved onto CUDA applications in the work of Jain et al. [31].

The solution involves splitting the process into two parts; an upper and lower half corresponding to the application level and the kernel level. The upper half takes care of the application logic, while the lower half communicates with the GPU. During operation, all calls made to the GPU are logged. The lower half is discarded at checkpointing time, and the log is written to a file. The upper half does not depend on external devices and thus can be checkpointed. When restoring, the lower half is first restored by replaying the logged calls, after which the upper half is restored with the C/R package. Memory pointers are resolved via a translation layer that connects the newly created pointers with the pointers in the log, assuming a deterministic context creation. This technique is called log-replay.

This technique also works for graphical applications, as seen in other research by the Nafchi et al. [32] and Hou et al. [33]. Both DMTCP and CRIU are tested for graphical applications interfacing the GPU with OpenGL enabling checkpointing with a log-prune-replay approach. The pruning step in log-prune-replay removes all logged calls not needed to restore the GPU state, such as destroyed objects. This keeps the size of the call log minimized.

The graphical output of the application was sent to a VNC server to encompass the X window state. It uses VirtualGL to enable hardware acceleration, but this approach is not suitable for cloud gaming as VirtualGL is not capable of hardware-accelerated video compression.

The research mentioned earlier illustrates a solution that is well optimized with low overhead and quick restore times. This performance results from the split-process approach, which carefully separates the two applications in memory space, while still being a single process. This technique is based on a process in process technique by Hori et al. [34]. This means that the upper half could directly call functions in the lower half and vice versa. However, this process is quite complex and not well documented; it requires careful function hooking and a tedious memory allocation management to discard the lower half at the time of checkpointing completely. Furthermore, it requires writing custom program loaders and linkers, which is nontrivial as these loaded programs fork when executing. Application restoring mechanisms must also be invoked from the lower half. It was not clear how a program such as CRIU, which incorporates many complex mechanisms, is loaded.

An interesting approach would be to separate processes. One process acts as the upper half with application logic and the other as the lower half interfacing with libraries. This is less complex

as the memory regions are naturally separated in processes. Complex memory discarding is also unnecessary as only the communication channel has to be disconnected and reconnected during C/R operations. This approach is inspired by the work of Zandy et al. [35] which portrays communication with so-called proxy processes via Remote Procedure Call (RPC). The intercommunication between the upper and lower half would consist of Graphics APIs instructions, similar to the streaming of graphics commands elaborated in the work of Eisert and Fechteler [2]

This technique was also mentioned in the work of Jain et al. [31], but allegedly suffered from too much overhead. This overhead is expected as RPC involves serializing instructions and sending them over a (local) network. Since only local interprocess communication (IPC) is needed, the two processes can also communicate via shared memory instead of RPC. IPC via shared memory requires is faster than RPC[14]. Furthermore, a two-process approach does not require a VNC server as function calls directed to windowing systems such as X can also be logged and replayed.

Migration

The work of Wang et al. [22] provides a survey on service migration, most specifically for mobile edge computing. A relocating mobile user would benefit from service migration when edge locations become out of reach and users enter a new service area.

Such a system specific to mobile gaming is Talaria, demonstrated by Braud et al. [36]. Talaria is an in-engine content synchronization solution for unnoticeable game instance migration between edge servers. Migration only leads to a service downtime below 25 ms, with a total migration time of 87 ms, making it suitable for high-performance gameplay. However, the C/R technique enabling migration is on application-level and thus not transparent. The game engine was adapted to facilitate the checkpoint and restoration of the video game.

Furthermore, in the work of Braud et al. a comparison is made with the research by Lin et al. [37]. Migration times are in the order of seconds, but the system provides a solution transparent to the video game. This work was executed with VMs rather than containers and shows an exceptionally high restoring time for simple OpenGL contexts. The work is over a decade old, and, given the improvement in C/R technologies, a revised work would pose a better comparison to the recent Talaria paper.

Considering strategy, we recall the work of Keni et al. [28]. In a scheduling environment, migration can be incorporated as an action incurring costs; the height of the associated cost will determine if the system benefits from saving costs by migrating at a particular configuration and incoming demand. The scheduling algorithm is solved twice in the respective work, with and without assuming that existing applications can be migrated and the more favorable action will be taken.

2.6. Conclusion

In this chapter, we have explored the pieces that make up a Cloud Gaming platform. An understanding of how video games work and which hardware is needed, gives way to understanding what solutions cloud computing and virtualization bring. Most of these cloud and virtualization technologies serve a different purpose than utilization with video gaming, which makes it interesting to apply these tools to cloud gaming.

GPU concurrency has been researched for general purpose computation, but information lacks when it comes to graphical workloads. Understanding more about the Graphics Pipeline and taking into account what resource restriction tools have been demonstrated for non-graphics APIs, explains that we need a method of monitoring and limiting calls made to the GPU. Container orchestration technologies such as Docker Swarm typically apply to replicated micro-service, and are thus in their general form not suited. Since we want to lower costs per user, a custom scheduling algorithm should be implemented to achieve this. This custom scheduler will be put to the test to see if the adaptations are worthwhile.

With Checkpoint and Restore, we could take this one step further by migrating video game instances to restore a tight-packed system. Multiple works have demonstrated a log and replay approach that is suitable for any type of application interfacing with the GPU, as well as video games.

[14] https://www.geeksforgeeks.org/difference-between-shared-memory-model-and-message-passing-model-in-ipc/

3

Design

With an understanding of the problems within multi-tenancy, scheduling, Checkpoint and Restore, and migration, we provide solutions with the design of Rapture in this chapter. We first go over the requirements that we have set to Rapture in 3.1. Then we will explain the virtualization technologies that have been implemented to run video game instances concurrently in Section 3.2. Next, an in-depth view of the algorithmic strategies for both efficient scheduling and migration are given in Section 3.3. Finally, we present a solution for enabling transparent Checkpoint and Restore of graphical hardware accelerated workloads, such as video games in Section 3.4.

3.1. Requirements

Before setting up the system, we want to pose requirements in order to validate the final system. This is divided into **Functional requirements** which are rules to abide to. **Non-functional requirements** are on a preferential basis, in other words, the system must strive to achieve these requirements in the best way possible.

3.1.1. Functional requirements

FR1 **Able to run hardware-accelerated graphical applications.** Hardware acceleration is essential to running video games. This means that Rapture as a cloud gaming platform should provide host machines in a cluster that are equipped with GPUs. Furthermore, the environment in which the video game is executed must allow communication with the GPU via graphical APIs such as OpenGL or Vulkan in order for the game to set up and execute a graphical pipeline.

FR2 **Allow for a multi-tenant environment, with stable virtualization.** An efficient cloud gaming platform utilizes host machines to their extent by allocating resources for multiple users on as few machines as possible. Cloud providers offer powerful hardware-accelerated computing in their data centers, including graphics cards with over 40 Gigabytes of video memory. Rapture should allow sharing of these hardware resources to fully pick the fruits of this available computing power.

This cohabitation of a single machine or node must be done with sufficient isolation and security. Much so that the user does not notice it is sharing physical resources with other users in this multi-tenancy environment. Stable virtualization will provide a user experience that is constant over time and does not change when other users join or vacate the respective host machine.

FR3 **Schedule instances on nodes tightly, creating high utilization of running nodes.** When a user requests a game instance for gameplay, there must be resources allocated in order to do so. The resource allocation must be intuitive, as we want to maintain a tightly packed configuration as much as possible on one machine, but do not want resources per user drained by cramming. A resource and application aware system should provide data to a scheduling system that ensures no such thing will happen. An occurrence of such an event is a violation to the agreed Quality of Experience (QoE) which should not be harmed.

FR4 Allow for migrating instances. Migrating allows for maintaining a tight-packed configuration while down-scaling cluster resources. This will drive down costs even more as overprovisioning is restricted.

Migration also opens up possibilities for fault tolerance and maintenance as machines showing peculiar behavior can be evacuated and examined. And within the edge-computing paradigm, it is also crucial for migrating between edge-locations in the case of a mobile user vacating a particular edge-location region and entering a different region with a more proximate edge location.

3.1.2. Non-functional requirements

NFR1 Transparency to the video game. There are a lot of video games that have been released for years, but are still popular to this day. Rapture should be able to run as many video game titles as possible. This should contain games developed for different operating systems such as Windows, different graphical APIs, and games developed in different game engines such as Unity[1]. There could be edge cases where a video game needs particular resources or is developed in a certain way that it requires adaptations to the platform, but we must strive to construct the system in such a way that many different out-of-the-box games will run on the platform.

NFR2 Satisfactory Quality of Experience. Cloud gaming should pose a viable alternative to normal gaming. This means that while resources are restricted for video games, the resources that are left provide a gaming experience competitive to today's standards. An acceptable resolution and frames per second should achieve this, and this should be the case for different video games while being constant throughout gaming sessions.

NFR3 On-demand game instance launching. A benefit of cloud gaming is that it enables on-demand gaming. Rapture should let the user decide at any time when to launch an instance and which video game to launch, with no prior planning or allocation requests. The platform should respond by answering this launch request by allocating the desired video game instance promptly.

3.2. Virtualization

Sharing hardware resources in Rapture is done with various virtualization techniques. These impose restrictions on the resources that the game demands, in order to leave resources for other games on the same host machine. The set of techniques in this chapter provides a stable video game environment for every user.

3.2.1. Rapture base container

Rapture uses Docker for OS-level virtualization and to package the video game with its needed dependencies in a container. Docker was chosen as it natively supports GPUs, and the early development of Rapture showed promising results in setting up a video game environment.

For packaging a Linux video game with Rapture, we use a base container[2] which is based on the nvidia/vulkan container. This container implements CUDA tooling, OpenGL, and Vulkan graphical APIs. The Rapture base container furthermore incorporates Vulkan tools[3], packages for X11 and the X window system, and an adapted version of the Libstrangle frame rate limiter[4].

For Windows video games, the Rapture base container is extended with the Wine compatibility layer and the DXVK extension that allows for running Direct3D Windows executable games in the container.

The Rapture base container can be used in Docker to create dedicated game containers from local or online game repositories as shown in the Dockerfile below.

[1] `https://unity.com/`
[2] `hub.docker.com/erwinrussel`
[3] `https://github.com/KhronosGroup/Vulkan-Tools.git`
[4] `https://github.com/ErwinRussel/libstrangle.git`

```
FROM  erwinrussel/rapturebase:linux

WORKDIR {/game}

COPY {local_game_directory} {game_directory}

WORKDIR {/game/game_directory}

CMD nohup bash -c "python3 /opt/strangle/promclient/promclient.py &" && \
    if [ $(xrandr | grep \* | cut -d' ' -f4) = ${RESOLUTION} ]; \
    then echo "Resolution already $RESOLUTION"; \
    else xrandr -s ${RESOLUTION}; fi &&
    strangle ${FPS} ./Game.x86_64
```

Here a game directory of e.g. a built Unity project is copied into the working directory of the container. Where on execution, a metrics daemon called Promclient is started in the background and the game is run under Libstrangle. The resolution and frames per second can be set from environment variables. Furthermore, the target display is also set by environment variables, as the scheduler must decide which virtual display to assign to the container.

This brings a hard dependency of this Rapture base container, as deployment requests binding to the X11 directory where an X socket is opened for the container to communicate with the display. In the case of a cloud gaming system, no physical display is available; thus a virtual display is started for the container to bind to. The GPU must allow for this, and a corresponding Xorg configuration must be set to expose a virtual display and allow for direct rendering.

In order to expose the GPUs to the Swarm cluster, the daemon configuration must be set accordingly setting the NVIDIA runtime with the target uuid of the GPU installed on the respective node.

3.2.2. Resource limitations

Docker provides ways to control how much memory and CPU a container can use. For memory, hard limits can be enforced that raise out of memory exceptions when violated or softer limits which allow the container to use more memory under certain conditions. Also memory can be swapped to disk if more memory is needed that allocated to the container, but this obviously imposes overhead. CPU can be limited by specifying how much and many CPUs the container can use. This is linked to the number of CPU cycles of the Linux kernel CFS scheduler. Docker modifies the settings in the container's cgroup.

Limiting GPU usage

As discussed in the previous chapter, to limit GPU usage, we resort to space and time restrictions.

Limiting space is done by limiting the rendering resolution of the video game; limiting this resolution decreases the size of the frame buffer that is loaded into the video memory, effectively lowering video memory per video game instance. Xrandr is the resize and rotate extension for the X window system and is used for setting the resolution of the virtual X Display, which in the case of Rapture is set to 1920x1080 or HD resolution. Further space limiting can be done by changing the rendering settings of the video game. A lower quality of textures and geometries require less video memory, but these settings are often only accessible in-game.

Limiting time is done with a framerate limiter; Rapture uses Libstrangle to set the desired framerate. Libstrangle hooks into the swap buffer call of different graphical APIs, which handles writing the new frame buffer as the last step of the graphical pipeline. This call is delayed by Libstrangle to lower the invocations per second to the desired number of frames per second. The result is a "gap" in graphical computation that can be filled by other applications or video games requesting graphical computation, time-sharing the graphical pipeline.

The duration of buffer swap delay is determined by the sleep time; this is how long the application should wait to adhere to the target-frame time, which is the planned time for swapping buffers in a limited scenario. This sleep time is adjusted by calculating the overhead of each frame, which is an average of how long it overslept, while not meeting the target-frame time. This overhead is subtracted from the sleep time and finally determines the idle period of the application. Furthermore, Libstrangle

calculates the actual framerate by looking at the elapsed time between function invocations, the potentially achievable and not used framerate calculated by overhead and sleep time, and the actual frame calculation time determined by the adjusted sleep time and achieved frames per second. This actual frame calculation time is essential to determining GPU time resource demands for a particular game.

Privacy and security
To create a secure system, we must make sure we do not expose data from other users. Sharing an X server with multiple user-allocated video game containers implies security considerations as a malicious user could obtain display and window information of other users. A solution to this security problem is running a single X server per gaming container and assigning the container to the display of that X server. More specifically, running a container under a UNIX user and using the `xhost +` unix command to grant access to a particular X socket prohibits other containers from switching display sockets in the mounted X11 socket folder. This, however, imposes some extra overhead as every X server requires little resources from the GPU.

3.3. Scheduler

The scheduler in Rapture utilizes metrics from the system and the game instances to decide where to place newly demanded game instances. The objective of the scheduler is to minimize the number of machines utilized for a set number of games. This is because a GPU is an expensive resource. Furthermore, the energy consumption of a GPU is linearly dependent on workload; there are no optimal curves for peak computation per wattage.

System overview
The scheduler of Rapture interfaces with Docker Swarm to place container workloads on a cluster of machines. These respective machines all feature a GPU and the necessary drivers for the GPUs and Docker to communicate with these GPUs, exposing them to the Swarm cluster. A network filesystem (NFS) is used to pass data between nodes, which is necessary for the migration of containers. Furthermore, one or more X displays should be running on the respective node for the container to bind to.

Deployed as a service to the Swarm cluster, the scheduler is a Python script that uses the Docker SDK to place game containers as services on the Swarm cluster. Next to the scheduler, some other persistent services on the Swarm cluster use an overlay network to communicate with each other. A Prometheus client runs as a single service on one of the nodes to collect metrics that the scheduler queries in its strategy. Prometheus is an open-source system monitoring and alerting toolkit that allows for scraping metrics from different services. These metrics can also be monitored manually via a deployed Grafana dashboard, which is a a web interface that can graph and display queries to the Prometheus client. For Prometheus to scrape services, one needs metrics exporters. The exporters used in Rapture are the following:

- **Node exporter** The node exporter from the Prometheus repository is deployed as a global service on every node. It exposes system metrics from the nodes in the Docker Swarm cluster such as disk I/O, filesystem, and CPU usage, load and memory statistics, and network info such as bytes transferred. This service is used to monitor the utilization of each node on a system level and see if CPU or memory utilization is exhausted.
- **cAdvisor** cAdvisor from Google provides resource usage and performance characteristics of running containers on the system. cAdvisor is deployed as a global service on every node. On a container level, cAdvisor checks resource utilization and provides a good insight on resources used by processes within the containers.
- **NVSMI exporter** Specific for NVIDIA GPUs, the NVSMI exporter reads metrics from the GPU. We use a custom variant of the exporter as the original needed to bind to the NVIDIA device folders. Unfortunately, binding to devices is not allowed in Docker Swarm mode; therefore, a new container with the NVSMI exporter was based on the nvidia/cuda container, which can interface with the GPU without needing the device mounts. Exported metrics contain video memory usage and GPU utilization, as well as power consumption and clock speed info. The container is deployed as a global service on every node.

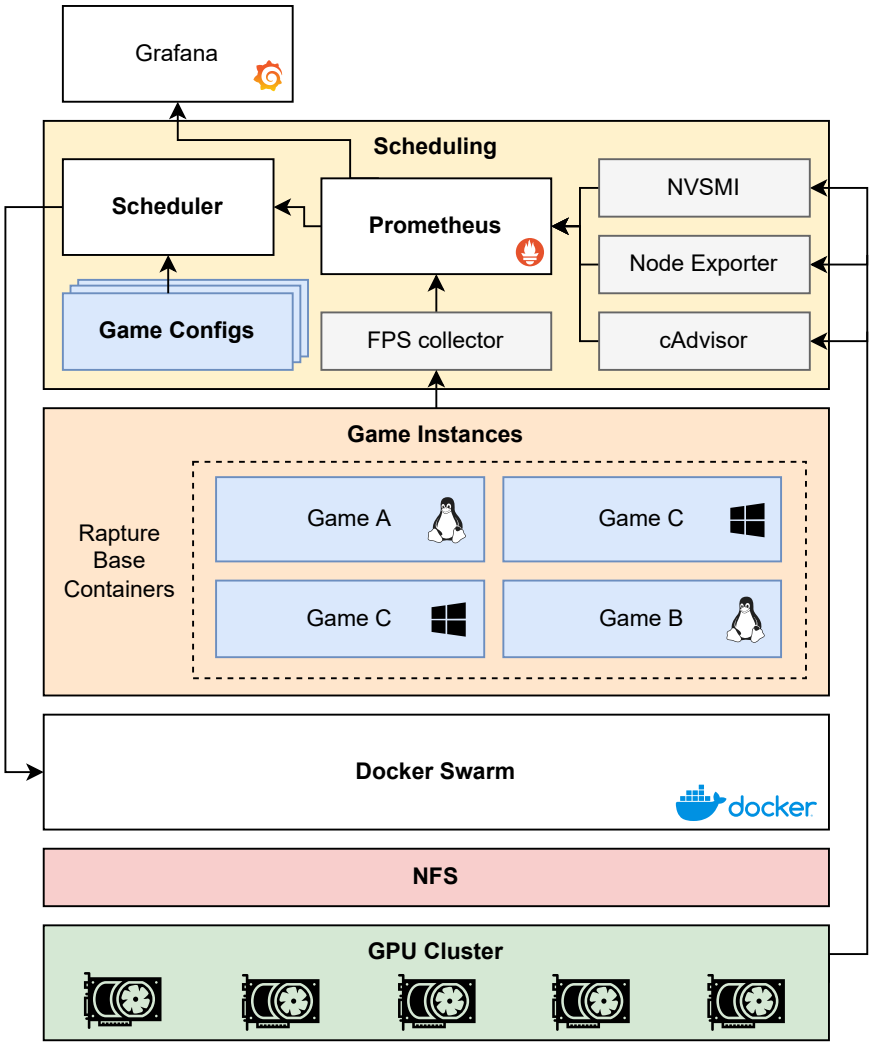


Figure 3.1: System overview

- **FPS collector** The game instances of Rapture incorporate a framerate limiter based on Libstrangle. The metrics of the framerate limiter are pushed to the Flask server of the FPS collecter via HTTP POST requests. This is needed because the game instances are not persistent like the exporter services are. The FPS collector collects these metrics, which are labeled per container and node. This labeling allows for querying framerate metrics per container or aggregating them on a node or cluster level. In turn, the FPS collector exposes these labeled metrics to be scraped by the Prometheus client. Periodically, the FPS collector checks which game instances have not updated their framerate data for a set amount of time. The metric corresponding to the container and node label is removed. The FPS collector is deployed as a single service on one node.

The scheduler can respond to incoming requests for gaming instances to be scheduled. For scheduling, the current metrics are pulled from Prometheus, and the profile of the to-be-scheduled game is taken from a game configuration file. This configuration file contains the container image and environment variables. Furthermore, it includes the amount of CPU, memory, video memory, and frame overhead times (based on a fixed resolution like 1920x1080). Finally, it includes the folder mounts for the X window endpoint and checkpoint files, and which ports must be opened for monitoring and streaming.

Strategy

Docker Swarm incorporates the spread strategy for deploying services; however, with constraints given to the services, this behavior can be adapted and even forced by constraining which specific nodes the service needs to run on. Docker natively supports the option to schedule with memory and CPU reservations and limitations not to exceed CPU usage on a node or have out-of-memory issues caused by another service or the service itself. This alone is not enough, as it will still schedule the nodes in a spread fashion, and it does not incorporate space and time reservations for the GPU.

A strategy based on the well-known greedy bin packing strategy seems most suitable to minimize the number of machines utilized for a set number of games. Other than a spread strategy, which achieves redundancy for replicated applications within a cluster, we aim to have a tightly packed system with high utilization numbers. In this way, the average costs of running a machine are divided over as many different users. Moreover, relatively more machines are left turned off to save costs. It must be noted that it is always good practice to keep a buffer of idle nodes readily available for increasing demand.

The Rapture scheduler takes in the following resource metrics per node to determine the best suitable node for scheduling a newly demanded gaming instance:

CPU Free

$$CPU_{free} = CPU_{tot} - CPU_{utl}$$

We use CPU halves determined by CFS seconds; since nodes can have multiple vCPUs in a cloud cluster, this number can exceed 1. The free CPU is equal to the total CPU of the node minus the CPU already allocated or utilized.

Memory Free

$$Mem_{free} = Mem_{tot} - Mem_{utl}$$

The free memory is equal to the total memory of the node minus the memory already allocated or utilized in bytes.

Video/GPU Memory Free

$$Vmem_{free} = Vmem_{tot} - Vmem_{utl}$$

The free video memory is equal to the total video memory of the node's GPU minus the video memory already allocated or utilized in bytes.

Average Adjusted Frame Sleep Time

$$FSleep_{avg} = min(avg(FSleep_{adj}), 10^9)$$

The average adjusted frame sleep time is a metric aggregated from the current game instances. This is the average time in nanoseconds of games sleeping per frame due to framerate limiting. It is a metric for computation time left in the nodes' GPU bandwidth. If no game instances are running on the node, the metric will output a sleep time of 1 second (one million nanoseconds).

With these metrics, we deploy a scheduling algorithm listed below. A reserve value from the config file specifies the resources needed for the specific game instance to run stably for each of the metrics.

Algorithm 1 Rapture Scheduler

procedure RaptureSchedule(g, N)
  for n in N **do**
    if $FSleep_{avg} < Ftime_{res,g}$ **then**
      $N \leftarrow N \setminus n$
      continue
    end if
    if $CPU_{free} < CPU_{res,g}$ **then**
      $N \leftarrow N \setminus n$
      continue
    end if
    if $Vmem_{free} < Vmem_{res,g}$ **then**
      $N \leftarrow N \setminus n$
      continue
    end if
    if $Mem_{free} < Mem_{res,g}$ **then**
      $N \leftarrow N \setminus n$
      continue
    end if
  end for
  if $N = \emptyset$ **then**
    Return
  end if
  $Sort(N)$ by Mem_{free}
  $Sort(N)$ by $Vmem_{free}$
  $Sort(N)$ by CPU_{free}
  $Sort(N)$ by $FSleep_{avg}$
  Deploy(g, $N(0)$)

The algorithm finds the best-fit for scheduling a game instance. It first eliminates nodes by determining if there are sufficient game resources (e.g. $CPU_{res,g}$) available to reserve for $FSleepavg$, CPU_{free}, $Vmem_{free}$, and Mem_{free}. The leftover nodes are sorted increasingly by free memory, free video memory, free CPU, and average frame sleep time. This sorting order is assumed to be from less exhaustive to the most exhaustive resource. A standard greedy bin packing strategy picks the first available node, which might not be the optimal choice. After obtaining a set of suitable nodes, the Rapture algorithm picks the node with the highest utilization to achieve a set of full-packed nodes with high utilization. The label of this specific node is set as a constraint when deploying a game instance to the Swarm cluster, at which Docker schedules it on the desired node. If no node is found, the scheduler does not deploy a game instance on any node, this means the cluster has reached its maximum capacity and should scale up by adding more nodes.

3.3.1. Migration

As we discussed before in the problem statement, migration can be a solution to create an even more tightly packed system in a case where the utilization per machine is low. The leftover idle nodes can be shut down, scaling down the cluster to save cost.

When migration is set to take place, the to be migrated game instance is sent a checkpoint signal and a folder location on the NFS filesystem. Then, the game is checkpointed, and the checkpoint files are written to this folder. A new game instance with a similar image is scheduled, with an environment variable for the game instance to know it has to restore from a checkpoint. After the restoration is done, the checkpointed game instance is removed, and migration is complete.

Strategy

Migration can be in effect when the scheduler recognizes that one or more nodes can be freed and shut down to save cost when migrating the game instances on that node. A threshold value can configure the aggressiveness of this behavior as these actions should be performed wisely, given the fact it is

noticeable to the user. Furthermore, there can always be an influx of new users that can fill the utilization gaps in the current cluster configuration. The strategy is listed below:

Algorithm 2 Rapture Migration

procedure RaptureMigrateListen(N, G, α)
  while True **do**
    $N_{sum,utl} \leftarrow 0$
    for n in N_{active} **do**
      $N_{sum,utl} \leftarrow N_{sum,utl}$+UtilScore($n$)
    end for
    if $avg(N_{sum,utl}) \leq \alpha$ **then**
      $Sort(N)$ by $utilscore$
      Migrate($N(0), N, G$)
    end if
  end while
end procedure

procedure UtilScore(n)
  $score \leftarrow 0$
  $score \leftarrow max(score, Mem_{utl,n})$
  $score \leftarrow max(score, Vmem_{utl,n})$
  $score \leftarrow max(score, CPU_{utl,n})$
  $F_{util} \leftarrow (10^9 - Fsleep_{avg,n})/10^9$
  $score \leftarrow max(score, F_{util})$
  Return $score$
end procedure

procedure Migrate(n_{evac}, N, G)
  for g in $G[n_{evac}]$ **do**
    RaptureSchedule(g, $N \setminus n_{evac}$)
    Remove(g, n_{evac})
  end for
  $N \leftarrow N \setminus n_{evac}$
end procedure=0

The strategy is divided into three functions, the listener loop, the utilization scoring, and the migrate function. The migrate listener periodically checks the utilization of each node by calling the utilization scoring function on each of the non-idle nodes. Suppose the average utilization of the nodes is below a threshold α. In that case, the nodes are sorted increasingly by their utilization score, and the node with the lowest utilization score is evacuated. The UtilScore function returns the max utilization between CPU, memory, video memory, and framerate utilization for each of the nodes. The framerate utilization is a function of the average adjusted frame sleep time to determine how much the GPU bandwidth is utilized. The max utilization score is returned to give a summarized overview of the general utilization of the nodes. The migration function uses the previously shown scheduling strategy to reschedule each game instance on the lasting nodes. The evacuated node is removed from the set of nodes and could be shut down as no game instances are left on the machine.

Given a specific migration threshold, one can ask when this migration will happen. This can be right after the threshold is met; however, it could interfere with the QoE of the gamer. A more user-friendly way of migrating instances recognizes which users are in intensive game-play, currently in combat in a shooting game, or which are in non-intensive game-play. Non-intensive game-play ranges from currently being in a menu or lobby or simply playing a game that does not involve quick response time. Furthermore, in a mobile setting, migration should occur from one edge-location to another when the streaming latency to a different location is less than the current latency. In this research, we do not go further in-depth on the timing of migration.

3.4. Checkpoint/Restore

To solve game migration, one must first solve checkpointing and restoring games. Checkpointing a Game is hard because checkpointing and restoring GPU state is nontrivial. Restoring state changes memory locations. A solution is logging the calls made to the GPU during execution. When a checkpoint occurs, the log is written to a file. Upon restoration, these calls are replayed to restore the GPU state on another machine.

In the background section, we have explored multiple ways to do this for a variety of applications (MPI, CUDA workloads); these researches were based on a split-process approach. Most of these researches lack an in-depth explanation of how memory management is executed when checkpointing and restoring,and involves a complex process. Upon initial experimentation, it showed that Checkpoint/Restore in User Space (CRIU) was the most mature and flexible Checkpoint and Restore solution.

Next to carefully managing memory, the split-process approach needs adaptation to work with CRIU. Since CRIU maps all memory regions in userspace for a single process, it tries to map the ”lower” part of the application that cannot be checkpointed as it has open device files that CRIU does not know how to checkpoint. There should be a pre-checkpoint routine that discards all of this memory before checkpointing, or CRIU should be adapted to ignore specific memory maps in checkpointing.

When restoring, the lower half of the application first needs to be fully restored. Then in the same fashion, the upper half must be reloaded into memory. We do not load the executable at this point, but one should load a CRIU restore process. From the specifications of the split-process in [33], it was not apparent how the lower half is preserved while the CRIU restore process is called (typically, CRIU restore executes as an `exec` command that replaces the entire process). The machine code, data, heap, and stack of the process are replaced by those of the new program. Next to that, it forks processes to restore the process tree; since this mechanism works in a single process space, it is not trivial how to combat these problems.

A clearer solution could be achieved by using a proxy process, which divides one process into two separate processes where one handles the application logic, and the other handles external devices and windowing systems. This is analogous to the two parts of the split process approach, with an upper half containing the application logic and the lower half acting in kernel space. We will use this terminology also in our solution.

These two processes communicate via a communication channel, which naturally incurs overhead. Next to sending commands over a channel, one must also serialize non-supported data structures. The difference with the split process approach is that the two processes (in this case, the application with the hook and upper half, and the lower half server that forwards these calls to the graphics libraries) are loaded into a single application, thus effectively being one process. This means that these two parts of application logic have direct access to each other’s memory, eliminating any overhead of serializing and sending these requests over a network. Isolation is obtained by strictly managing memory and allocating enough space for the programs to not interfere with each other (for instance, a growing heap of the first ”process” could override the memory of the second ”process”).

3.4.1. Proxy process via IPC

The solution we present decouples an application into upper (logical) and lower (driver) parts and has these parts communicate via Interprocess Communication (IPC). This can be seen in Figure 3.3. In order to efficiently create a suitable IPC communication channel, we use a small function generation script. Like the `rpcgen` command for Remote Procedure Call (RPC), the script generates the upper half shared library and lower half executable with the corresponding header file after providing a list of functions. A hook library is generated, which hooks and overrides the provided functions by preloading this as a shared library into the application. This hook library is a separate file from the upper half, which calls the corresponding IPC functions in the upper half. This way, if an application is developed from scratch, it could also directly target the IPC functions in the shared upper half library.

IPC via shared memory

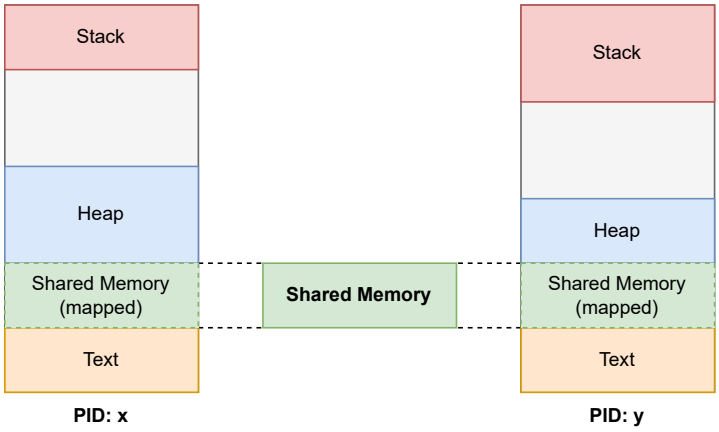


Figure 3.2: Shared memory between two processes

We implement this communication channel by opening a shared memory location between the two processes. This shared memory location features a message block written to by the upper and lower half, each taking turns reading and writing. This is a synchronous single-threaded communication strategy that uses simple synchronization mechanisms in order to have the lower half listening for requests and the upper half listening for responses after a request has been submitted. IPC via shared memory is also faster than RPC, as only system calls are made for sharing the established memory, and the networking capabilities of RPC are not needed.

One can compare this implementation to a serving hatch in a restaurant. Where the upper half is the dining area with its guests and waiters, and the lower half is the kitchen that prepares the food. When the guest has ordered, the waiter places the written order in the serving hatch for the chef to collect and execute. After the chef makes this food, he places the food in the serving hatch for the waiter to serve to the customers. In this serving hatch, the function requests (order) and responses (food) are dropped in the shared memory space of the two applications.

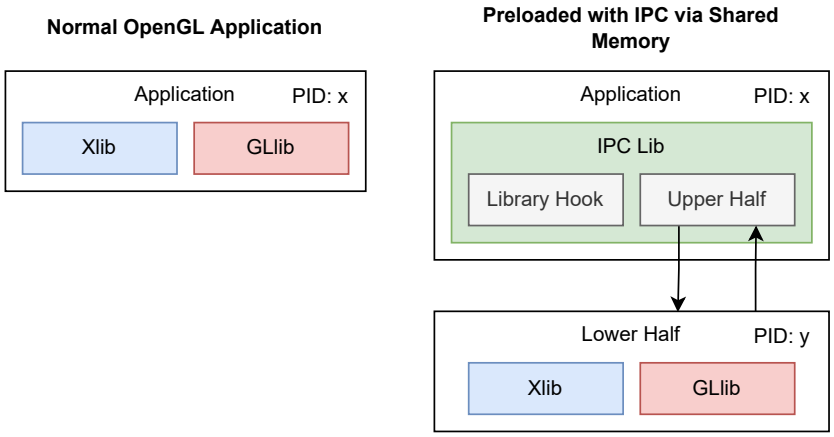


Figure 3.3: Difference of normal application versus Preloaded with IPC via Shared Memory library

Finding the set of functions to be hooked

As discussed before, we have one domain of the application that we are splitting into the upper and lower half. These halves must be carefully curated to the application; the slightest slip results in a segmentation fault (often the result of pointers that are not available in the domain of the upper half, called by an unhooked function). To estimate which functions need to be hooked, one can use reverse-engineering tools such as `ldd` and `ltrace` to determine the libraries and library calls and `strace` to determine the system calls. For X window and OpenGL calls, Xtrace and Apitrace provide solutions to reverse-engineer the calls made to these APIs. However, a more universal solution would be to encompass the whole library of a windowing or graphical API so that every call is hooked and communicated

to the lower half.

3.4.2. Message Block and Serialization

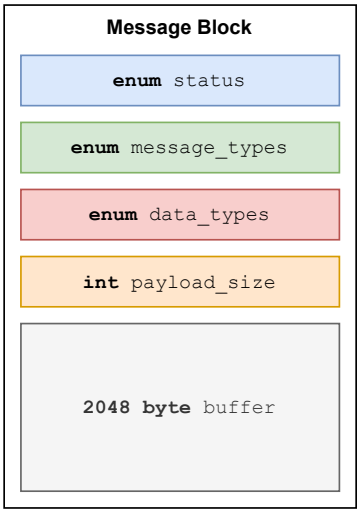


Figure 3.4: IPC message block

The message block that is read and written to in the shared memory space consists of a status variable, message headers, and the payload buffer.

- **Status.** The status enumeration variable synchronizes between the upper and the lower half. Initially, this integer is set to LISTENING, which denotes the lower half-listening to new function calls by the upper half. The upper half writes the function request to the message block and sets this status variable to CALL. The lower half reads the function request, executes it, and writes the response to the message block. The status variable is set to RESPONSE, at which the upper half reads the response variable and continues execution, setting the status variable back to LISTENING.

- **Message types.** The message type variable is either CALL or RESPONSE, which denotes that the message is a function call or a function response and the direction of communication. CALL is from the upper half to the lower half and RESPONSE is from the lower half to the upper half.

- **Data types.** Given a RESPONSE message type, the data type variable denotes which data structure is sent via the message block. This could be as simple as an Integer or a Float but encompasses every data type that is normally used in common virtual address space. With a CALL message type, the data type variable corresponds to a function that is called by the upper half. This function often has one or more inputs, and for each of these functions, a tailored struct with these input data types is sent via the message channel.

- **Payload size.** The payload size is the size in bytes of the sent data type or function call struct. This payload size is used to check if the payload size corresponds to the data type and is further used to determine the size of the memory copy in reading. When writing to the buffer, the payload size is set to the number of bytes written.

- **Buffer.** The buffer is a large block of bytes where the data types and structs are written to and read from via a memory copy. The size of the buffer must accommodate the largest serialized struct in the set of functions that are communicated. For other functions, this buffer is mainly unused as only the payload size is written and read from the buffer. The only implication is that the shared memory space is somewhat more significant, but it is not transmitted over a channel or network, as is the case with RPC.

Serialization
Since the upper and lower half consists of two separate memory spaces (except for the shared memory block), it is impossible to reference a variable or structure via pointers as these pointers live within the scope of a single address space. This means that data that is sent over the communication channel must never contain pointers and only values of the data structures.

With a shared header file that both the upper and lower half includes, a catalog is made of which data structures are known by both sides and can be communicated via the message block. We create a struct for every function input that incorporates the input variables of this function. When the upper half calls a specific function via the hook, the hook writes the input variables to this struct, whereafter the struct is written to the message block for the lower half to extract the input variables. The input variables are used to run the requested function, and the response data type is written to the buffer.

In a language such as C, serialization is especially important as data structures such as arrays are referenced by a pointer, pointing to the first element. This means the structs corresponding to functions that take in arrays or structs must be adapted to take in all of the elements of the array or structure and not the pointer.

For instance, if one has a function `int ADD_ONE(int a)` that increments the input value, then calling this function will return the input incremented input value. However, another way of getting the same result could be done with a void function that returns nothing. In this case, we have function `void ADD_one(int* a)` that does not take in the value, but the pointer of the number variable to be incremented. Calling this function will increment the value of the variable referenced by the pointer.

This means that the hook functions must transform a function that takes in a pointer into a function that gets the values of these pointers and writes these into structs. The data type of this value will also be the return type of the formerly void function. After returning to the client-side, the value will be set to the variable referenced by the pointer. This can be a very complex process, given that in a language such as C, one can have multiple layers of structs within structs, and these have to be serialized until a data type references a value.

3.4.3. Log and Replay

The decoupled application's lower half or proxy process cannot be checkpointed as it interfaces with the GPU via graphics APIs. Checkpoint and Restore packages do not incorporate saving the memory state of the GPU, and since GPU architectures and drivers are secretive, another approach is needed.

Usually, an application talks via a single graphics API with the GPU; thus, the GPU state is encompassed in the information passed via this API. This state can thus be restored by replaying the information. In this case, graphics calls to the point before the checkpoint.

Since the IPC via shared memory communication channel already incorporates serialization and writing the function calls and responses to a memory block, we extended this to write these calls to a binary file. At the point of restoring, before starting the function listener, the lower half first replays the calls from the binary file to restore the GPU state.

One thing to be very aware of is that memory addresses change for the recreated objects when replaying these calls. Functions that reference an object via a pointer must go via a translation layer, which translates the old pointers referenced by the upper half to the new pointers in the lower half. This translation layer is set when replaying the calls and incorporating the pointers returned by the graphical APIs with the pointers from the logs.

All the calls requested by the upper half and executed by the lower half will be logged. However, some of the function calls are not logged, as they are not suitable for replay. These function calls belong to the group of functions that are affected by external events, an example in Xlib is `XNextEvent()`. This function takes in a pointer and waits for the next event while interacting with the X window. The result of this interaction, i.e., the event type that has occurred, is generally processed by the application in the upper half and transmitted via the IPC communication channel by the lower half. Since, at checkpoint time, these events are processed and thus encapsulated in the state by the upper half, it is not necessary to replay these calls. Furthermore, while replaying these calls, the upper half is not restored yet, so there is no possibility of communicating with the upper half.

Finally, logged functions can become redundant. An object that is created and later destroyed does not have to be recreated and re-destroyed when restoring the GPU state, as the target state does not incorporate the existence of such object. An extension to the Log and Replay approach is a pruning daemon that listens for calls that destroy an object or context and backtraces the calls made on this

object or context. These calls can then be removed, and the log can remain a lightweight representation of the GPU state.

3.4.4. Container

While Docker has experimental support for CRIU, the adaptations to the video game executable into a checkpointable upper half and a non-checkpointable lower half mean this implementation is not useable. The reason is that (external) CRIU checkpoints all of the processes within the container and thus tries to checkpoint the lower half.

The solution is to install CRIU within the container (it must also be installed on the system to work) and call CRIU only on the upper half of the application. This is an extension of the previously elaborated Rapture base container. The checkpointing and restoring can be coordinated externally by the scheduler using the ‘exec‘ command of Docker to run a checkpointing script. Restoring is done via an executor script that runs the application normally or restores it from a checkpoint given the environment variables passed to the Rapture container.

This running script checks for the restore environment variable; if it is false, the script first starts the lower half executable, opening the shared memory block. Next, it preloads the application hook into the game executable, and communication between the upper and lower half commences. If the restore environment variable is set to true, the application starts the lower half executable with the ‘replay‘ flag and the file path to the log file. The application replays the log and, after this replay, opens the shared memory block and continues normally by listening for incoming calls. The upper half is restored by calling ‘criu restore‘ with the checkpoint folder location as an input argument. When CRIU restores the upper half, a regular operation is resumed. In both cases, the upper and lower half PIDs are written to a file such that the checkpointing script can utilize these.

The checkpointing script is simple; it takes the PIDs from the file and sends a SIGUSR1 signal to the upper half application. This upper half incorporates the CRIU C library, which means the process can call CRIU on itself. This step is crucial as CRIU does not allow checkpointing applications with a shared memory block. The application’s signal handler first detaches itself from the shared memory space before it calls CRIU on itself. When restoring this application, the application continues in this handler which reattaches the shared memory, and regular operation continues. The lower half is simply killed after the upper half has checkpointed, as the log is already written to the file while the application is running. The checkpoint script finally moves all the checkpointing files into a folder on the NFS filesystem. This means that the Docker container must bind a volume to the NFS filesystem on the host.

3.5. Conclusion

In this chapter, we have discussed the design decisions for Rapture. First, in 3.1 we formally defined what the functional and non-functional requirements are that the system needs to adhere to. Second, we discuss in section 3.2 different virtualization methods implemented in the system to create a stable multi-tenant environment. The scheduler and migration strategies and an overview of the system have been discussed in section 3.3. In section 3.4 we thoroughly went into the use of inter-process communication over a shared memory channel, to decouple the application in a lower and upper half. Where the upper half can be checkpointed and restored and the lower half logged and replayed

4

Experiments

Unless stated otherwise, experiments were conducted on the Google Cloud Compute Engine. The Machine features an n1-standard-4 (4 vCPUs, 15 GB memory) configuration coupled to a Tesla P4 Graphics Processing Unit with 8GB of VRAM. The machine is running Debian 10 Buster and experiments were conducted in containers running Docker 20.10.9. The NVIDIA Driver Version: 418.211.00 and CUDA Version: 10. this driver incorporates Vulkan support. Implementation of Rapture and IPC via shared memory can be found on Github[1]

4.1. Resource restriction on containers and GPU multi-tenancy

In order to test the performance of Rapture, we need a valid benchmark that will suffice as a simulation for a video game.

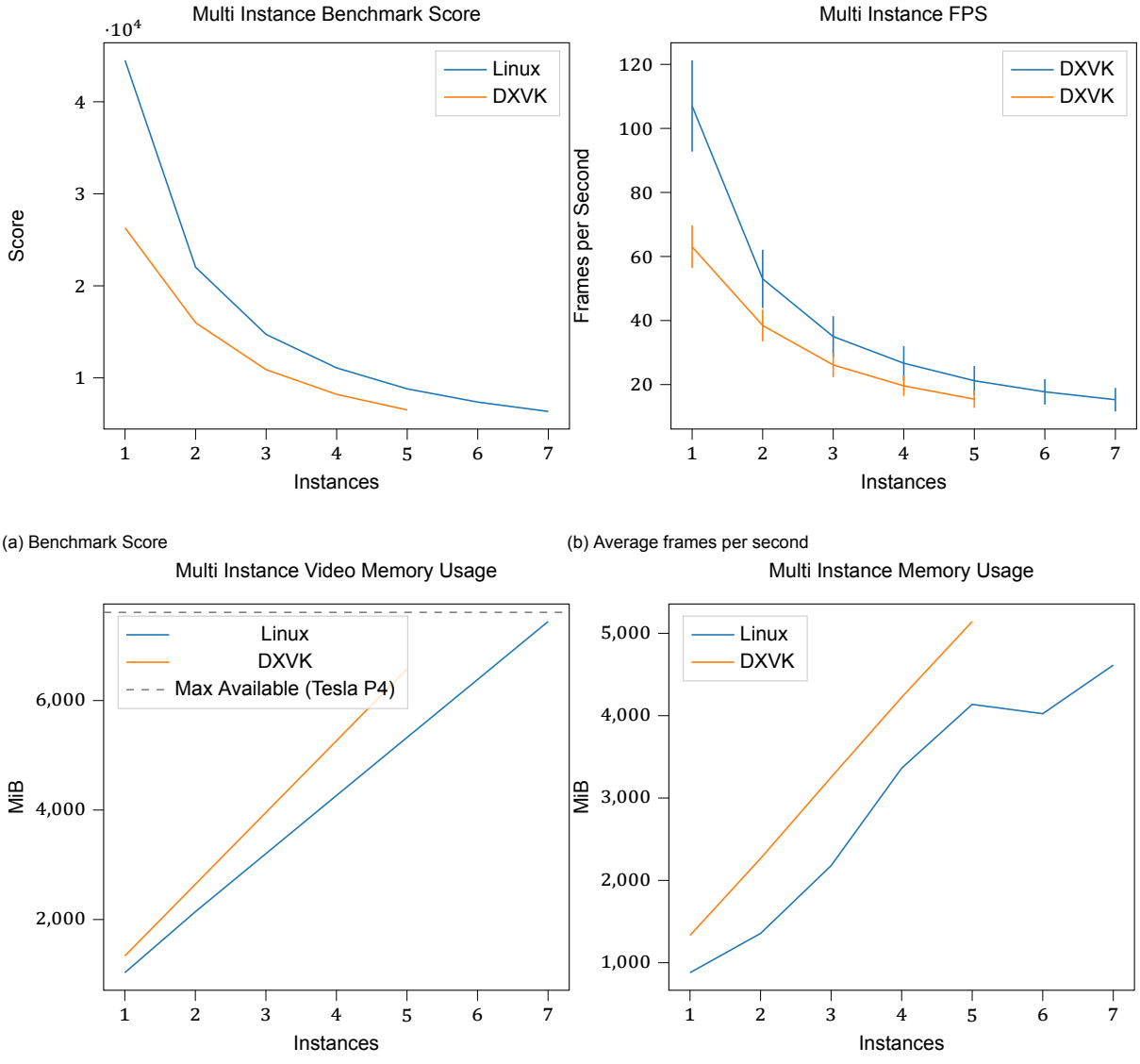
Figure 4.1: The Viking Village Benchmark

The Unity Viking Village project, is a model scene with different objects, shaders, textures and animations. The scene has a fly through camera that highlights different parts of the village and makes it suitable for benchmarking as no input is required by the user. Both a Linux and Windows version of the same benchmark has been built, to create a fair comparison between a native Linux build and a Windows build running on Wine with DXVK. The project has been adapted for benchmarking, as we

[1] `https://github.com/ErwinRussel/rapture`, `https://github.com/ErwinRussel/atlas`

are recording the frames per second and calculate the benchmark score (sum of all frames), average and standard deviation of the frame rate for determining stability. Furthermore, use the `nvidia-smi` and `top` command to get general information about the GPU and CPU

4.1.1. Experiment 1: Multiple game instances on a single host



(a) Benchmark Score

(b) Average frames per second

(c) CPU memory usage, lower is better

(d) GPU memory usage, lower is better

Figure 4.2: Multiple game instances on a single host

To first verify that multiple video game instances can run oncurrently and establish a baseline of how the Viking Village benchmark runs on the system, we ran different amount of instances at the same time and explored how many concurrently can be run and what effect it has on the benchmark. The benchmark was run with a resolution of 1920x1080.

From Figure 4.2, it is apparent that a native Linux build demands less resources than a Windows build running on Wine with DXVK. For the Linux build, the maximum amount of game instances was 7, while for the Windows build it was 5. The reason is that the video memory is earlier exhausted as the Windows build takes up more video memory.

Considering QoE, 30 frames per second should be acceptable for game play, which means that the Linux build can run 3 instances with this frame rate on the machine, whereas the Windows only allows

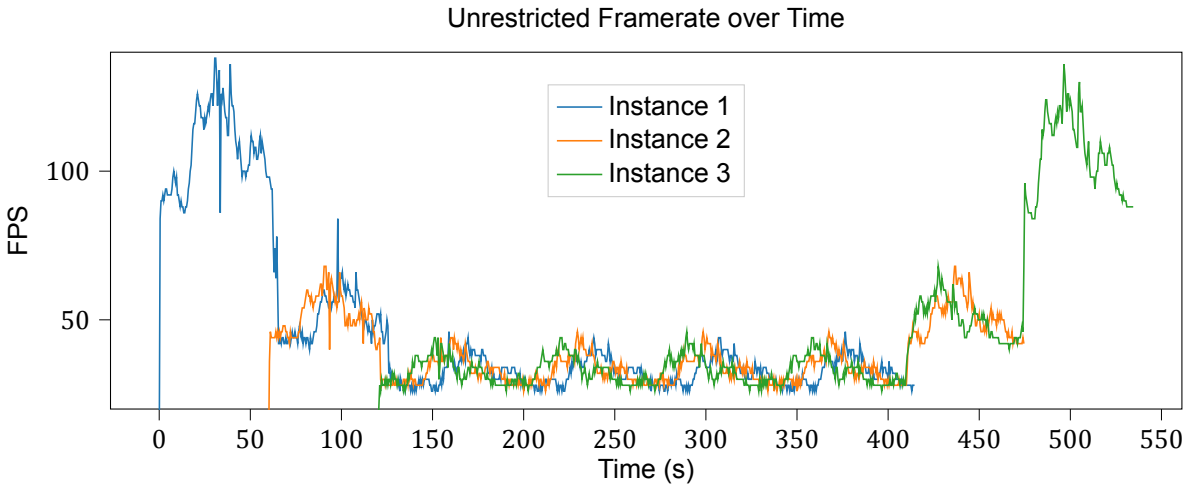


Figure 4.3: Frame rate over time for three game instances starting one after another.

for 2. Finally, the CPU usage is plotted, which is again higher with the Windows build. However, it is still on the low side: a CPU with 6 Gigabyte of memory is sufficient.

Frame rate over time
Another experiment to assess the user's perspective of an unrestricted multi-tenant environment can be seen in the Figure 4.3. In this experiment, three containers with the Linux build of Viking Village are launched, with a delay of 60 seconds between the video game instances. The first user benefits from high frame rates until the second and third user requests a game instance, and the frame rates are evenly divided between the users throughout the experiment. When the first two users leave, the last user benefits from high frame rates again. Interestingly, a sinusoidal frame rate deviation was observed between all three users throughout the experiment.

4.1.2. Experiment 2: Effect of resource restrictions

Now that we have seen the environment in an unrestricted setting, we isolate the restriction techniques and observe their effects. For both limiting time and space usage of the GPU we experiment with different frame rates with Libstrangle as a frame rate limiter, and different resolutions by setting the desired resolution with Xrandr.

Limiting Frame rate

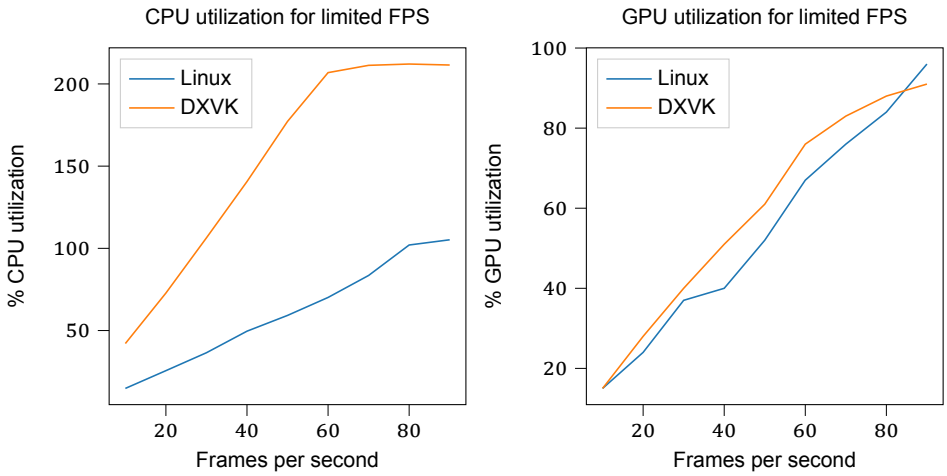


Figure 4.4: CPU and GPU utilization with different frame rate limitations.

With an uncapped frame rate, the video game will render as much images as the processor and graphics pipeline allows to provide the user with smooth gameplay. While the graphics pipeline is mostly the limiting factor, it is interesting to see what the CPU and GPU utilization is when the frame rate is limited. As the game idles to wait for the next image to draw, the process is not utilizing CPU and GPU and thus a lower frame rate means lower CPU and GPU usage. This can be clearly seen in the line graph in Figure 4.4. Where for the Linux build the CPU utilization grows linearly with the frame rate. For the Windows build on Wine with DXVK, we see that the CPU utilization is higher and hits a ceiling early on. The reason of the high CPU usage, is that the translation layer occupies CPU time as every Windows system and Direct3D call needs to be translated to Linux system and Vulkan calls. A value over 100% means that the application is using more than one core.

Limiting Resolution

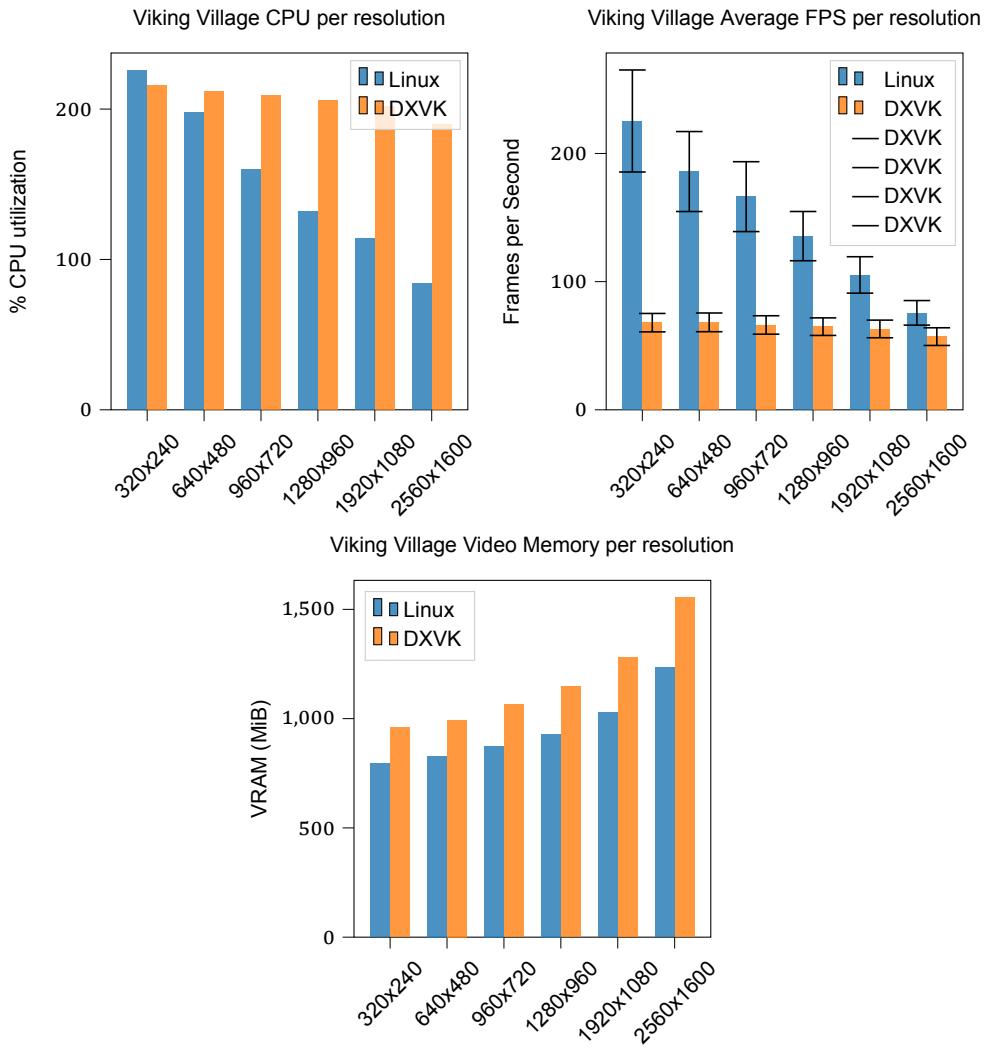
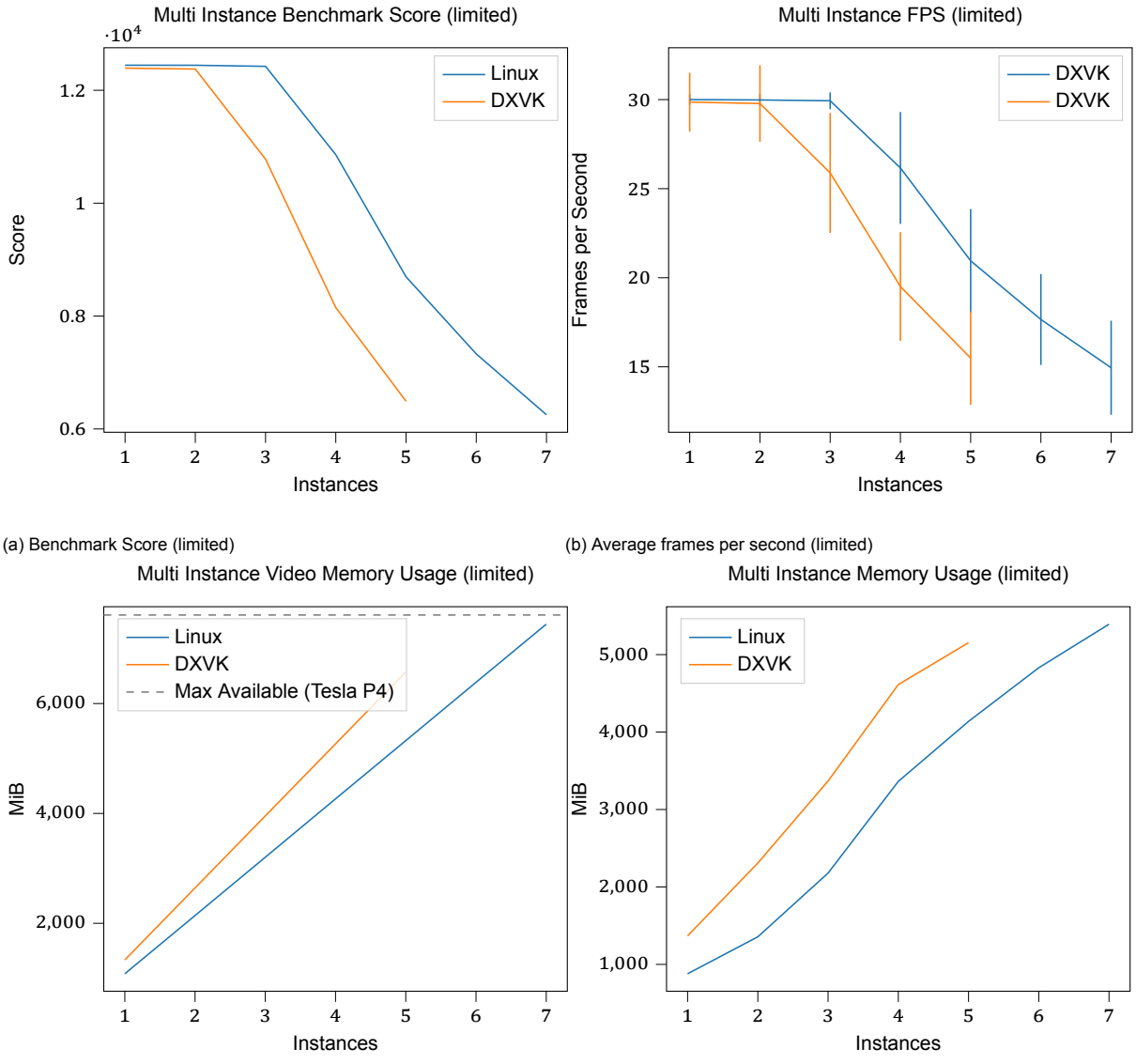


Figure 4.5: CPU utilization, frame rate and video memory utilization at different rendering resolutions.

With limiting resolution and uncapped frame rate, we see in Figure 4.5 that a lower resolution introduces a higher CPU utilization. This is most noticeable for the Linux build. The reason for this is that when the resolution is lower, the graphics pipeline has less pixels to calculate and draw on the frame buffer, which means there is a higher frame rate which also demands more CPU time. This higher frame rate with lower resolution can be seen in the top right figure. This also means that a lower resolution requires less GPU utilization or bandwidth for a fixed frame rate. Important is the lower usage of video memory with a lower resolution, as we saw in the first experiments, the number of concurrent video

game instances was exhausted by video memory. Limiting the resolution lowers the video memory and allows for more concurrent video games to be executed.

4.1.3. Experiment 3: Demonstrating Isolation by imposing resource restrictions



(a) Benchmark Score (limited)

(b) Average frames per second (limited)

(c) CPU memory usage (limited), lower is better

(d) GPU memory usage (limited), lower is better

Figure 4.6: Multiple game instances on a single host, frame rate limited

Now that we have shown the effectiveness of frame rate and resolution limiting of video game instances. We revisit Experiment 1 with frame rate limiting to 30 frames per second. In Figure 4.6, we see a clear roof of benchmark as this is made up by the sum of all frames rendered. Again, the Linux outperforms the Windows build and can run 3 instances with the same benchmark score. With more than 3 instances for Linux and more than 2 for Windows, the capped frame rate is not achieved, and naturally it begins to deteriorate.

Frame rate over time

The user perspective test in Figure 4.7 shows desired results, while the first and last user do not benefit from high frame rates at the beginning and end of the experiment, users entering and leaving do not

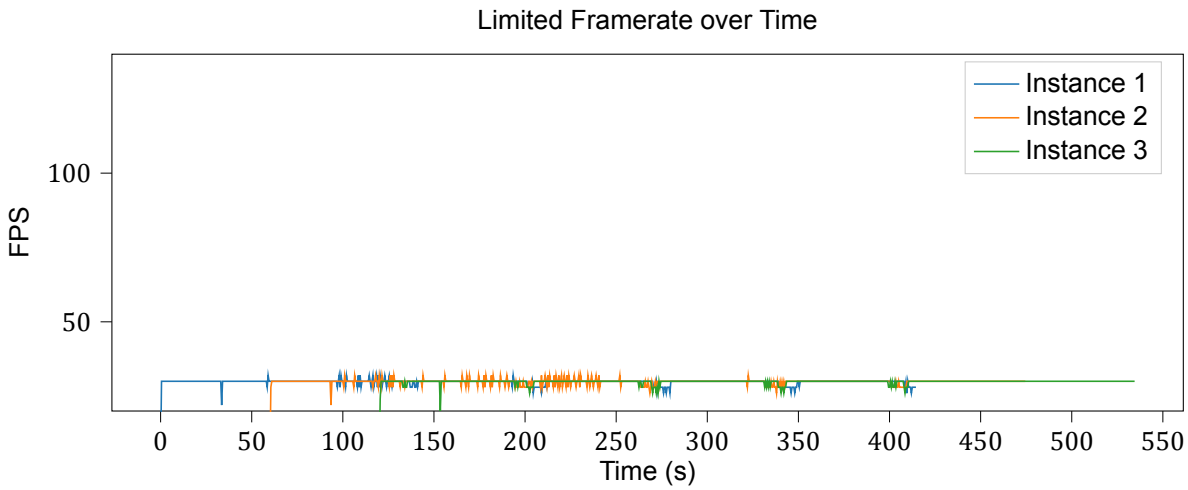


Figure 4.7: Frame rate over time for three game instances starting one after another, frame rate limited

change the frame rate of other users. During the experiment the frame rate is stable and therefore, we see that limiting the frames per second effectively achieve this notion of isolation.

4.2. Scheduler

Now that we have seen how we can achieve stable concurrency on a single system, we need an efficient scheduler for placing video game workloads on multiple nodes in a cluster. We use three of the same machines as have been used in the experiments before as GPU nodes. Furthermore, a separate node with 2 vCPUs serves as a workload manager and runs the scheduler and conducted experiment. This to make sure that every node has similar resources available when scheduling video game instances.

To assess the performance of our best-fit Rapture algorithm, we compare it to other well known scheduling algorithms. Latest versions of Docker Swarm only support scheduling services in a spread fashion, where each service is divided over the cluster nodes evenly, only legacy versions of Docker Swarm also incorporated a Binpack and Random algorithm. We will compare the best-fit Rapture strategy to Spread, Binpack and Random strategies.

A distinction between the Rapture strategy and the other strategies is that the Rapture strategy incorporates GPU and CPU demands, whereas the other strategies only require CPU demands. We did however set the CPU demands for the games to be generous, as to have a sufficient competitive advantage to compare with.

We have conducted four experiments of 20 minutes to measure the strategies' performance:

1. **High Load Random Requests** This experiment requested between 1 to 9 video game instances randomly, while also randomly descheduling video game instances as to simulate leaving users. 9 video game instances is the maximum the cluster can run stably, it is therefore for the strategies to determine the best placement as to have a balanced placement of video game instance.

2. **Low Load Random Requests** In this experiment, different to the previous experiment, the video game instances were limited to a maximum of 6. This experiment is in order to demonstrate the capability of the scheduling strategies to leave machines idle, as to save cost.

3. **High Load Sine Requests** A sinusoidal wave of requests from 0 to 9 video game instances, with video game instances randomly descheduling. This experiment was to see the step-by-step choices of the different scheduling algorithms.

4. **High Load Sine Requests** Similar as the previous experiment, only limited to 6 video game instances. Again to demonstrate the capability of the scheduling strategies to leave machines idle, as to save cost.

For these experiments we have measured a plethora of metrics. These consist of metrics that denote the amount of nodes used and the amount of video game instances on a particular node. We collect metrics that measure frame rate for containers and the violation of these frame rates per node. Finally, system utilization metrics are measured to give insight on the load on the machines. All these metrics per strategy and experiment can be found in Appendix A1. In this section we will be going over the amount of utilized nodes, as this is the objective we want to minimize with our best-fit Rapture scheduler. Furthermore, we will look at the number of containers violating frame rate requirements as this is a metric that shows deterioration of QoE of a cloud gaming platform.

4.2.1. Experiment 1: High Load Random Requests

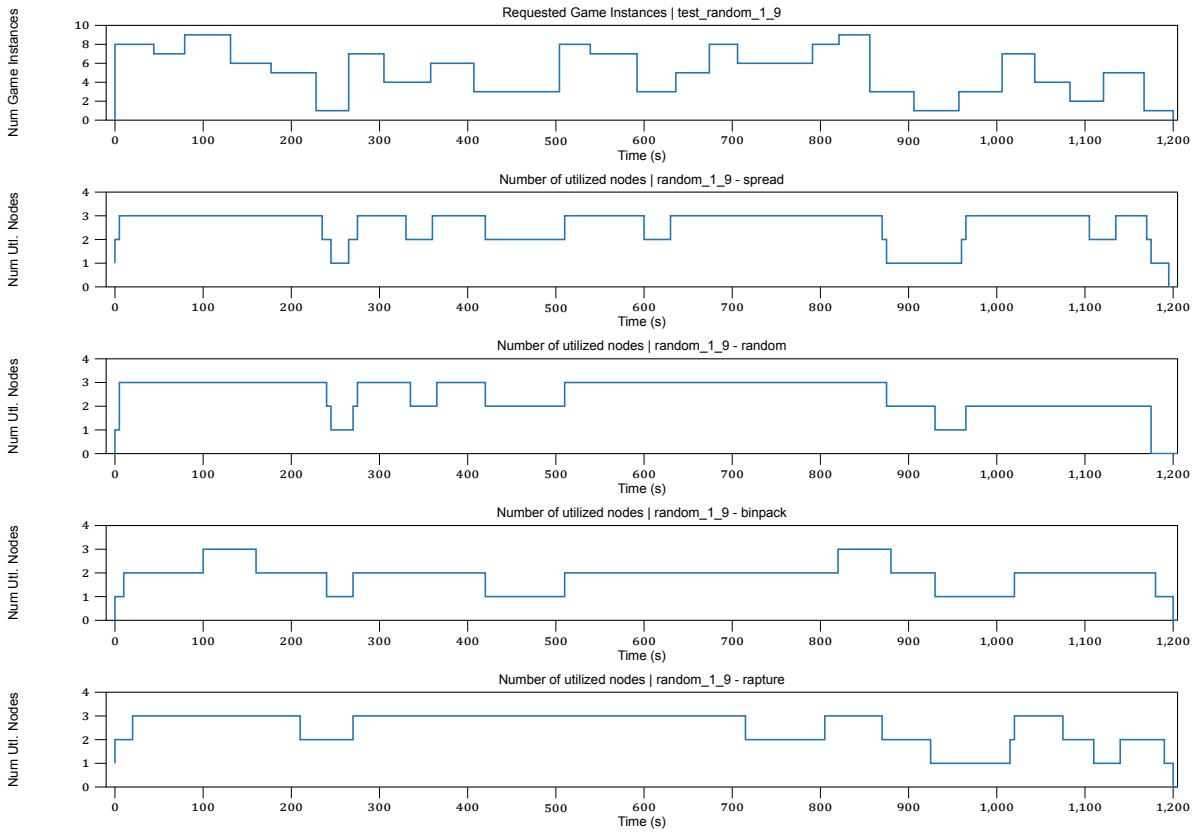


Figure 4.8: Node utilization for high load and random incoming requests

First looking at the number of utilized nodes for each strategy in Figure 4.8, with the requested game instances as the top graph in the figure above. We see that over the course of the experiment, Binpack showed the lowest utilization of nodes. For this experiment, the advantage of the Rapture scheduler is not prominent over the Spread scheduler, only at a later stage. Note that for different experiments users can leave at different nodes randomly, which has some variation in outcome. Furthermore, metrics from Prometheus have some delay and only scrape intervals of 5 seconds, and gather this information every 15 seconds. We also measure the area under the curve (AUC) score of the node utilization graph, which counts how many nodes are used per second. If 3 nodes are used during a 1200 second period, the AUC score is 3600 which is the maximum for this experiment. In Table 4.1 one can see the AUC scores and improvement over the maximum AUC score.

Table 4.1: Area under curve scores for node utilization

	Spread	**Random**	**Binpack**	**Rapture**
AUC score	3120	3000	2280	3030
Improvement	13.3%	16.7%	36.7%	15.8%

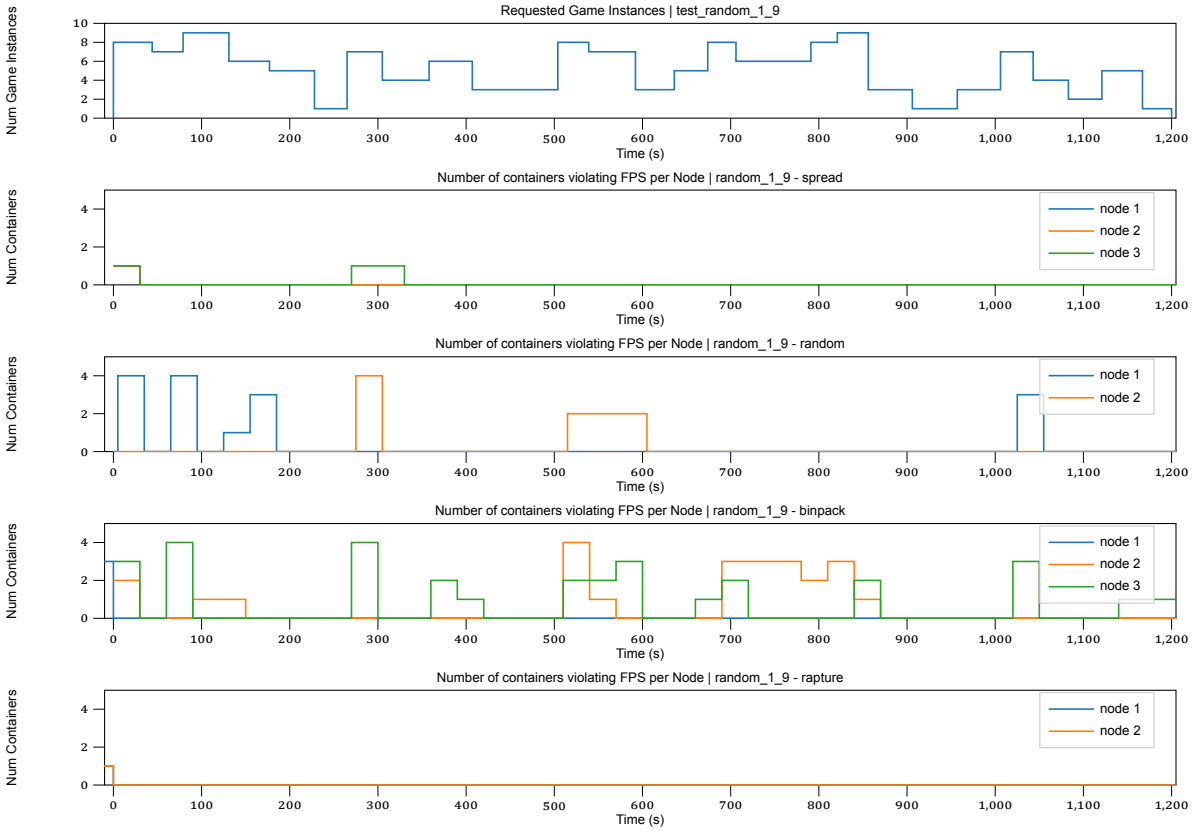


Figure 4.9: Frame rate violations for high load and random incoming requests

When looking at the number of containers violating frame rate per node in Figure 4.9, we see that the advantage of Binpack with low node utilization manifests itself in many violations of frame rate. This only for the second and third node, as the first node is largely unused in this strategy. Also the Random strategy incurs violations. The violations of Spread could be a wrong measurement or unforeseen mishap as the algorithm should evenly divide game instances. The Rapture best-fit algorithm showed no violation of frame rate.

4.2.2. Experiment 2: Low Load Random Requests

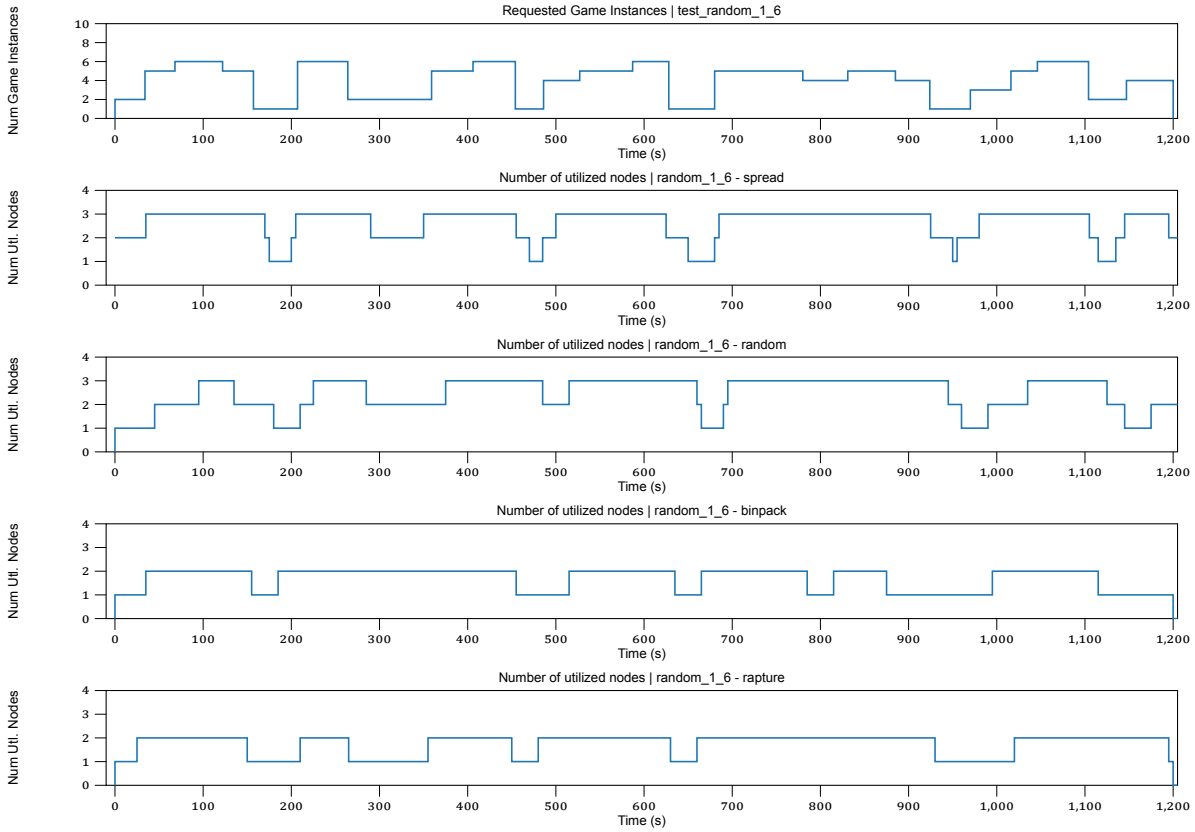


Figure 4.10: Node utilization for low load and random incoming requests

With a lower load, it is apparent in Figure 4.10 that both Binpack and the Rapture scheduling algorithms outperform Spread and Random utilization wise. Both Binpack and Rapture best-fit utilize only the two necessary nodes, whereas the other algorithms utilize all three, with Spread almost constantly using three nodes. In Table 4.2 we see that Binpack and Rapture have more than double the improvement of Spread and Random.

Table 4.2: Area under curve scores for node utilization

	Spread	**Random**	**Binpack**	**Rapture**
AUC score	3190	2945	2010	2070
Improvement	11.4%	18.2%	44.2%	42.5%

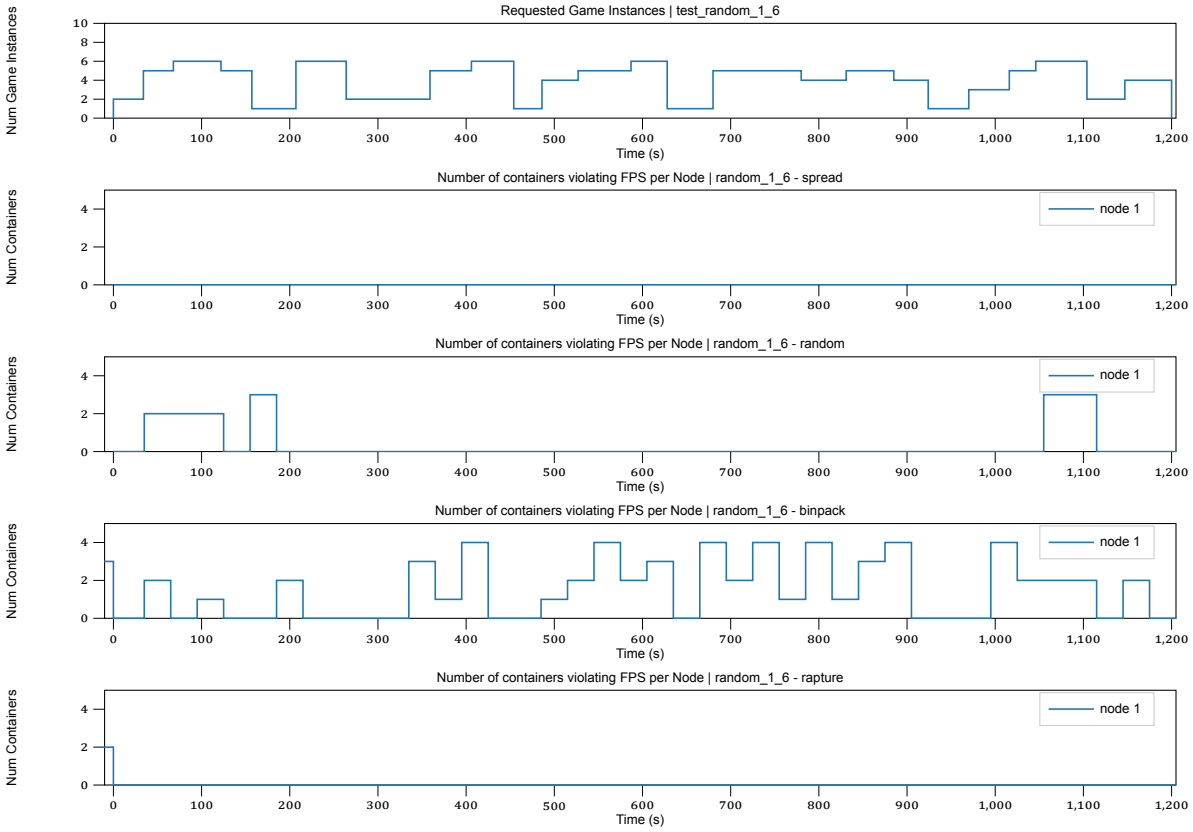


Figure 4.11: Frame rate violations for low load and random incoming requests

In Figure 4.11, we can see that with a lower load both Spread and Rapture best-fit are spotless. Binpack overpacked video game instances on the first node and Random made some undesired placement choices.

4.2.3. Experiment 3: High Load Sine Requests

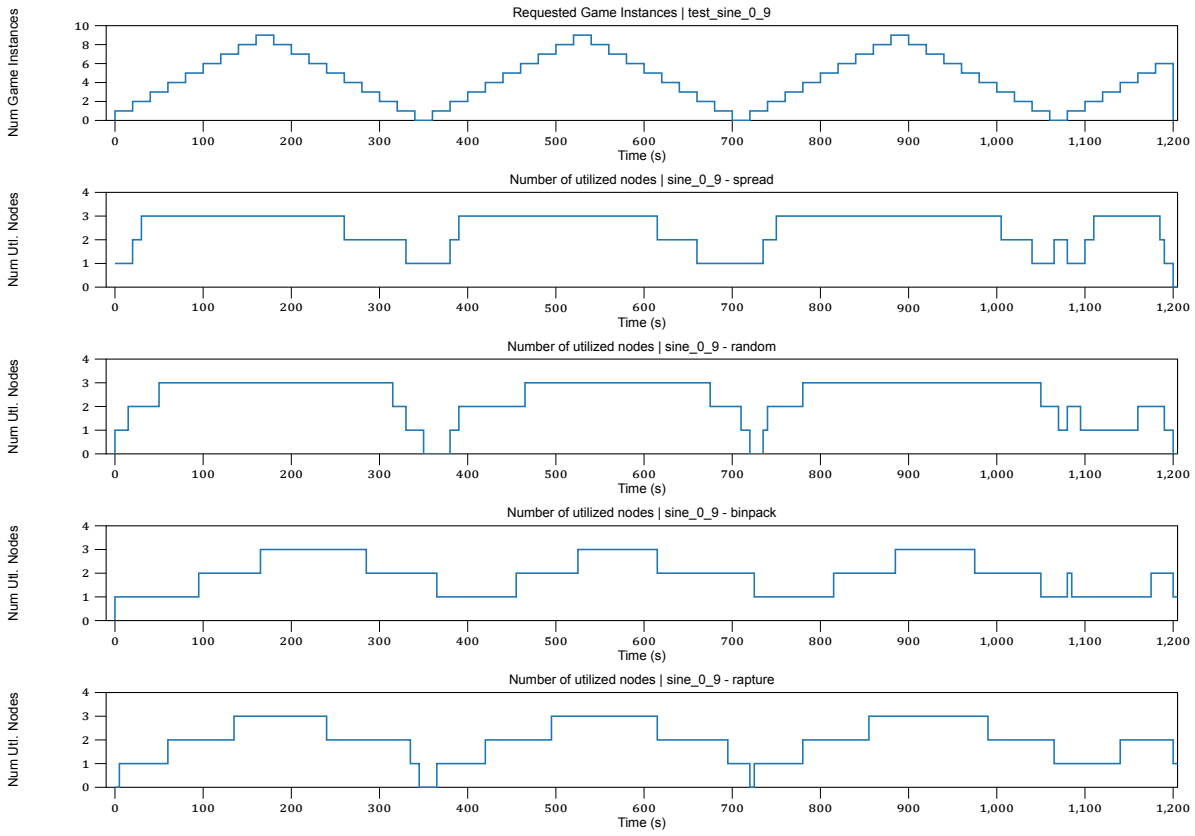


Figure 4.12: Node utilization for high load and sinusoidal incoming requests

With the sinusoidal demand of video game instances, it is more clear how these scheduling strategies behave and how optimal their choices are as shown in Figure 4.12. The algorithms Binpack and Rapture best-fit mimic the load in terms of nodes utilized whereas Spread and Random overprovision for the workload necessary. In Table 4.3 we can now clearly see that Rapture and Binpack achieve similar node utilization improvements.

Table 4.3: Area under curve scores for node utilization

	Spread	**Random**	**Binpack**	**Rapture**
AUC score	2990	2910	2310	2430
Improvement	16.9%	19.2%	35.8%	32.5%

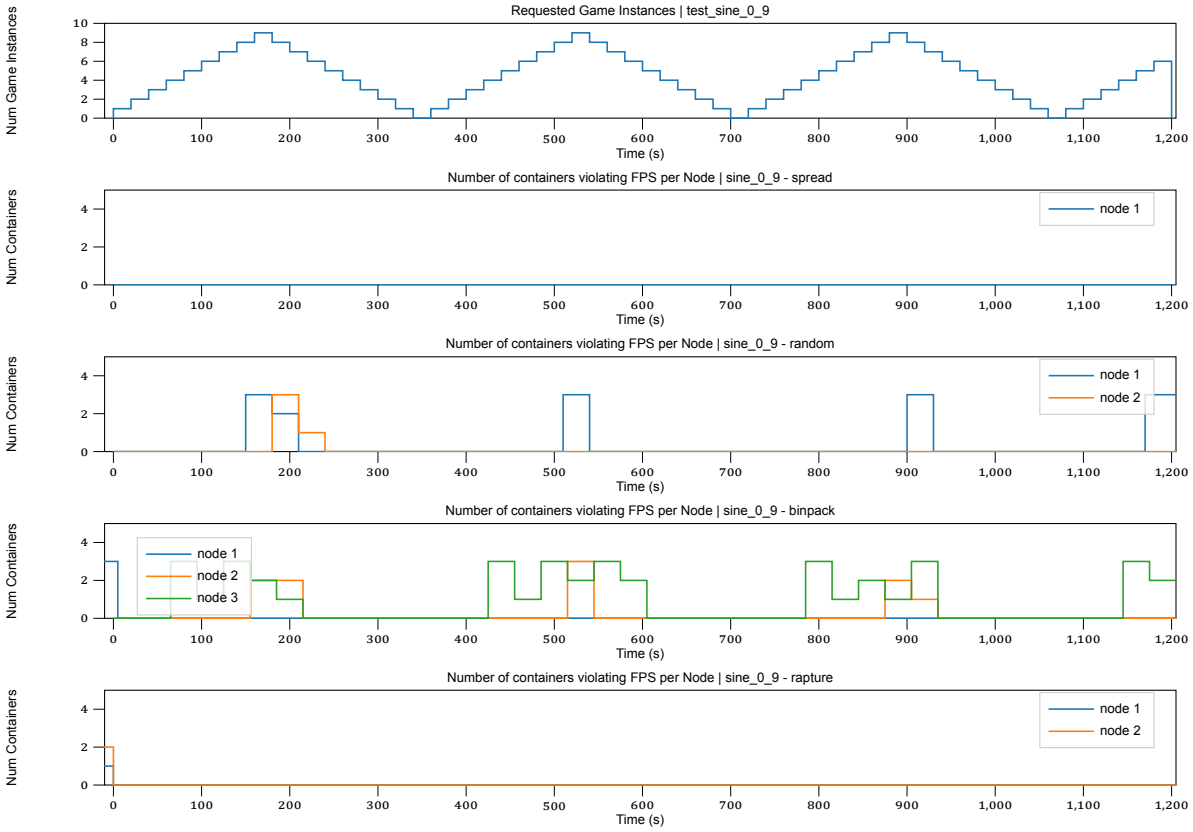


Figure 4.13: Frame rate violations for high load and sinusoidal incoming requests

For Binpack we see in Figure 4.13 that a high number of video game instances always result in violation of frame rate as it packs too many instances on the first available nodes. Random outperforms Binpack multiple times given frame rate violations. Spread and Rapture best-fit are both spotless.

4.2.4. Experiment 4: Low Load Sine Requests

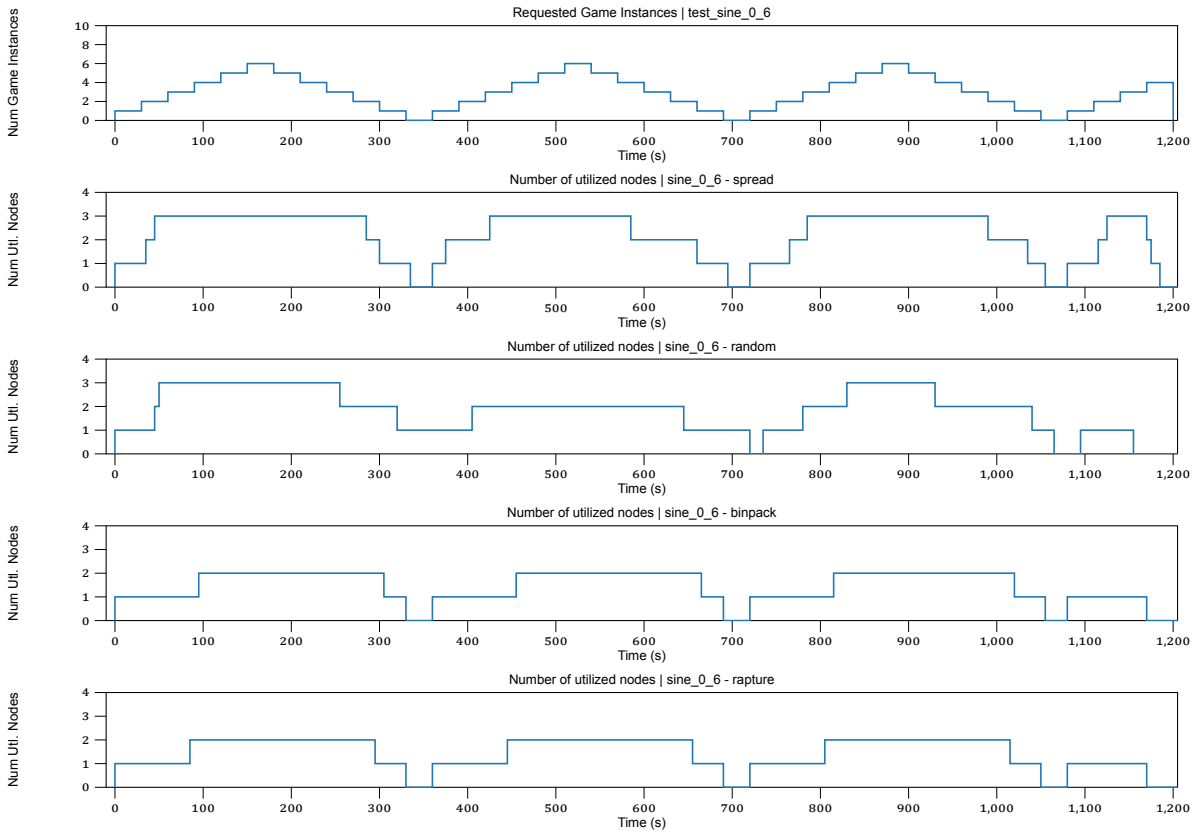


Figure 4.14: Node utilization for low load and sinusoidal incoming requests

Finally the low load sine requests show the same fallacy in Figure 4.14 for Spread and Random over utilizing the amount of nodes. Binpack and Rapture show an impressive improvement of 52.5% in Table 4.4, which means they have used half of the resources compared to three constant running nodes.

Table 4.4: Area under curve scores for node utilization

	Spread	**Random**	**Binpack**	**Rapture**
AUC score	2640	2190	1710	1710
Improvement	26.7%	39.2%	52.5%	52.5%

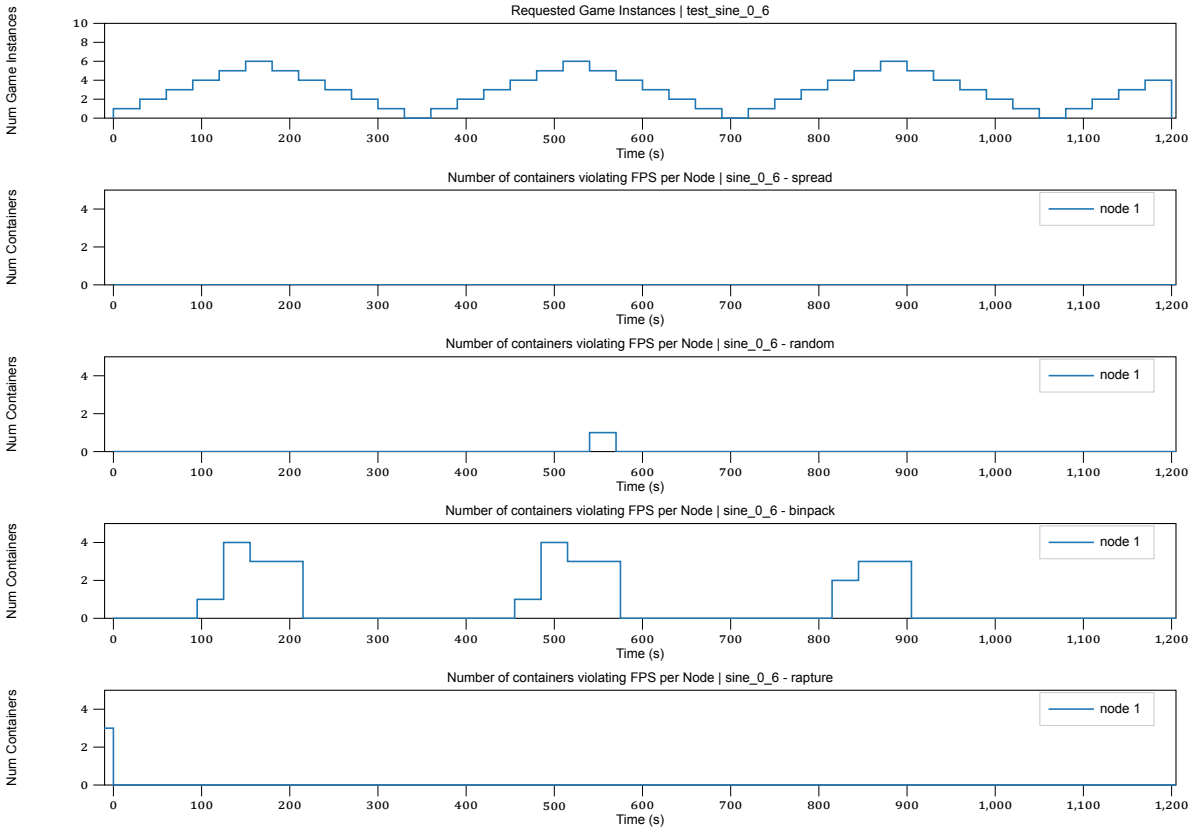


Figure 4.15: Frame rate violations for low load and sinusoidal incoming requests

In Figure 4.15 we see that Binpack showed the expected frame rate violations, while the rest of the algorithms showed good preservation of QoE.

From these experiments we see that the Rapture best-fit algorithm performs exceptionally well given preservation of QoE. For node utilization the Rapture best-fit algorithm mimics Binpack without the poor choice of over-packing on a single node. It outperforms the Spread algorithm that is native to Docker Swarm and confirms that building a separate scheduler can enhance efficiency in node utilization levels.

4.3. Checkpoint and Restore

In the design chapter, we posed a solution for enabling Checkpoint and Restore by decoupling the application into a lower and upper half that used IPC via shared memory. This method was implemented for the simple benchmarking application Glxgears, which is a small OpenGL model scene that tests frame rate of the machine.

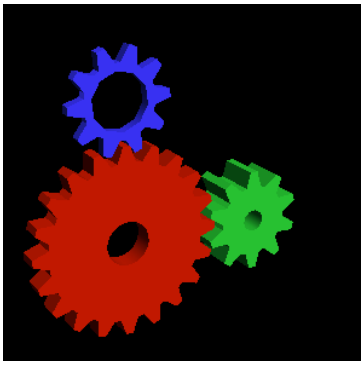


Figure 4.16: Glxgears benchmark

A subset of OpenGL calls was hooked and added to the IPC via shared memory setup, enough to run, checkpoint and restore the application. The implementation of the IPC via shared memory for Glxgears is 7365 lines of C code, a list of hooked functions can be found in Appendix B. Some small adaptations to the Glxgears application were made, as it incorporated a handful of function macros that cannot be hooked by preloading the IPC via shared memory library. For these macro's, similar OpenGL functions where used and hooked by the preloaded shared libary. This takes away from the transparency of the checkpointing solution.

To ensure that the log size of the lower half did not grow to large sizes, the lower half resetted the write pointer of the binary file on every GlxSwapBuffer call, as Glxgears builds up the initial state and later only rotates its components. Therefore, it is sufficient to only replay the first setup and last few calls for synchronization, in order to succesfully restore the lower half.

The following experiment was conducted on a VMWare Fusion virtual machine running Ubuntu 20.04, on a i7 Macbook Pro with 16GB of RAM and a Radeon Pro 560X 4GB GPU.

4.3.1. Experiment 1: Overhead of IPC via shared memory and logging calls

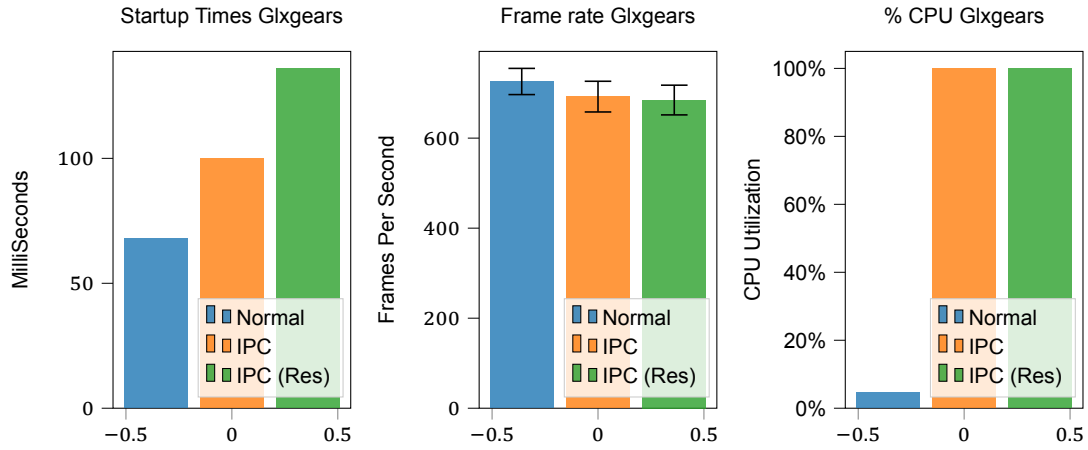


Figure 4.17: Startup times, Frame rate, and CPU utilization for normal Glxgears, Glxgears under IPC via shared memory, and Glxgears restored under IPC via shared memory.

We measured the startup times for Glxgears, without the IPC via shared memory preloaded, with, and restored from a CRIU dump and lower half log. The results are shown in Figure 4.17. We noticed that startup times are in the same order of magnitude of milliseconds, this is also attributed to the fact that there is not much state to restore for the restored Glxgears.

Suprisingly, uncapped frame rate was similar for both native Glxgears and Glxgears with IPC via shared memory. This similar frame rate is likely due to the low graphics performance of the VM. However, when capping frame rate to 30 frames per second, one can clearly see the immense overhead the IPC via shared memory solution has. The reason for this is a naive implementation of the upper and lower half synchronizing via status variables and listening to this variable with while loops. These while loops, like uncapped frame rate, request as many CPU seconds as possible.

4.4. Migration

With Checkpoint and Restore implemented for Glxgears, we initially wanted to containerize this setup in order to run on the three GPU nodes. The migrating sequence involves sending a checkpoint command to the to be migrated containers, at which the checkpoint files and lower half logs are written to a mounted volume that is connected to a NFS directory for all nodes. On the target host this directory is mounted by a newly scheduled game instance, at which it restores from this log.

Unfortunately, while separate Checkpoint and Restore worked, Docker Swarm does not allow for running privileged containers as root user. This capability is needed for CRIU to access the kernel. All of the code is in place, which even features ephemeral "proxy" services within Docker Swarm that send the checkpoint commands to the to be migrated game instances. This is needed as in the control plane of Docker Swarm, there is no option to directly run the "exec" command on a container running on another

host. If newer versions of Docker Swarm allow for running privileged containers, the experiment can be conducted as intended. For this specific experiment, Kubernetes would have been a better container orchestration framework. The argument is that Kubernetes does allow for privileged containers within pods, and sending commands to pods is possible via the ”pod exec” command.

It was chosen to conduct the experiment with the Linux build of Viking Village, and simply descheduling and rescheduling the video game instances. For experiments, we only considered the high load requests, as these have more range in demand and provide a good experiment to observe the migration behavior. We compared no migration with a migration threshold (Alpha) of 70% average utilization and 60% average utilization.

4.4.1. Experiment 1: High Load Random Requests

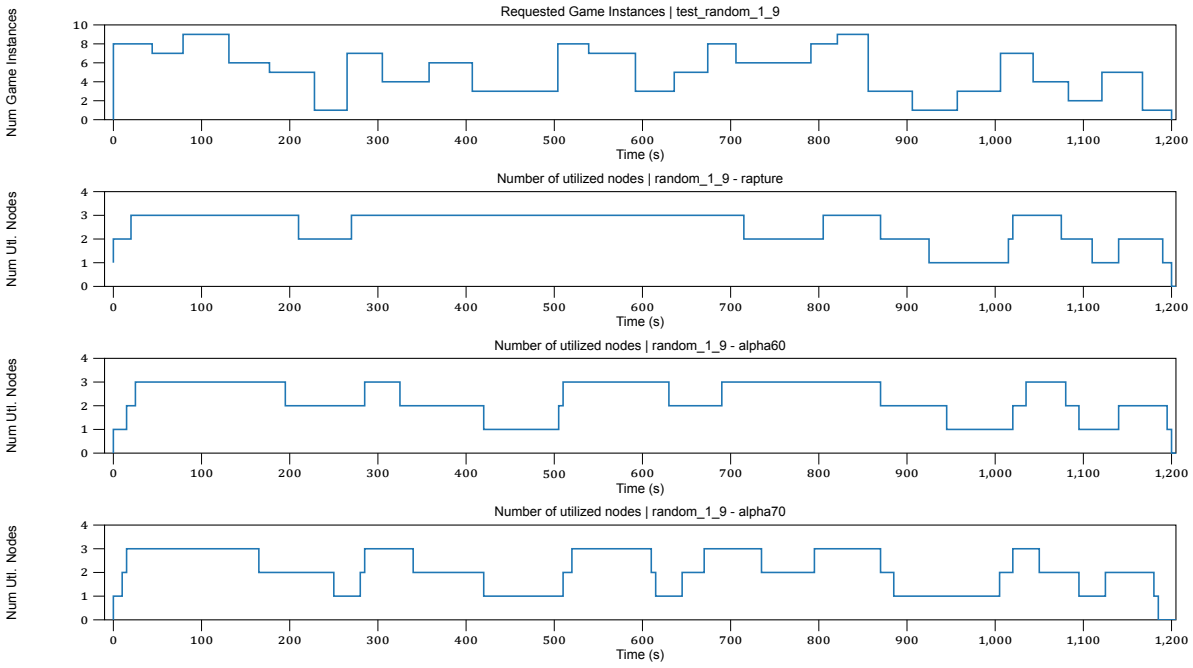


Figure 4.18: Node utilization for high load and random incoming requests, comparing migration disabled and migration enabled for utilization thresholds of 60% and 70%.

As expected, a higher utilization threshold instigates more aggressive migration behavior. We see in Figure 4.18, that comparing an Alpha of 70% and 60% to no migration, enabling migration allows for less node utilization throughout the experiment. In Table 4.5 we see that an Alpha of 70% shows almost a double improvement in node utilization over the Rapture best-fit algorithm without migration enabled.

Table 4.5: Area under curve scores for node utilization

	Rapture	**Alpha 60%**	**Alpha 70%**
AUC score	3030	2730	2520
Improvement	15.8%	24.2%	30.0%

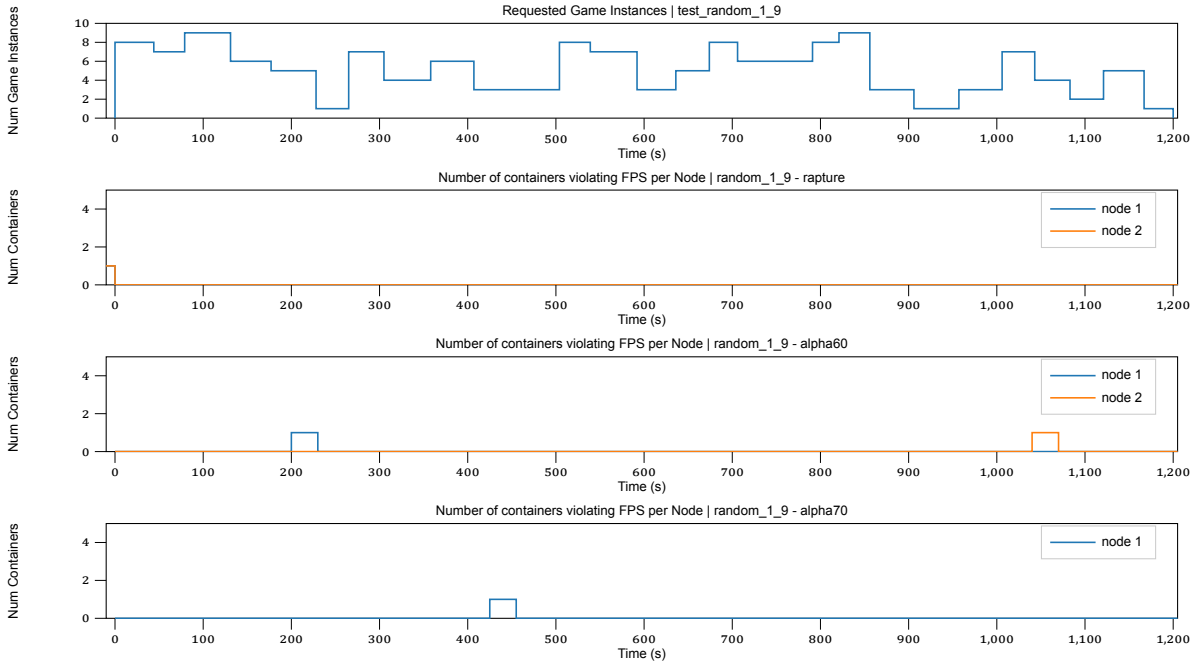


Figure 4.19: Frame rate violations for high load and random incoming requests, comparing migration disabled and migration enabled for utilization thresholds of 60% and 70%

Interestingly, we can see in Figure 4.19 that while migration deteriorates QoE as the user is disconnected for a short period, these experiments also showed some instability regarding containers violating frame rate.

4.4.2. Experiment 2: High Load Sine Requests

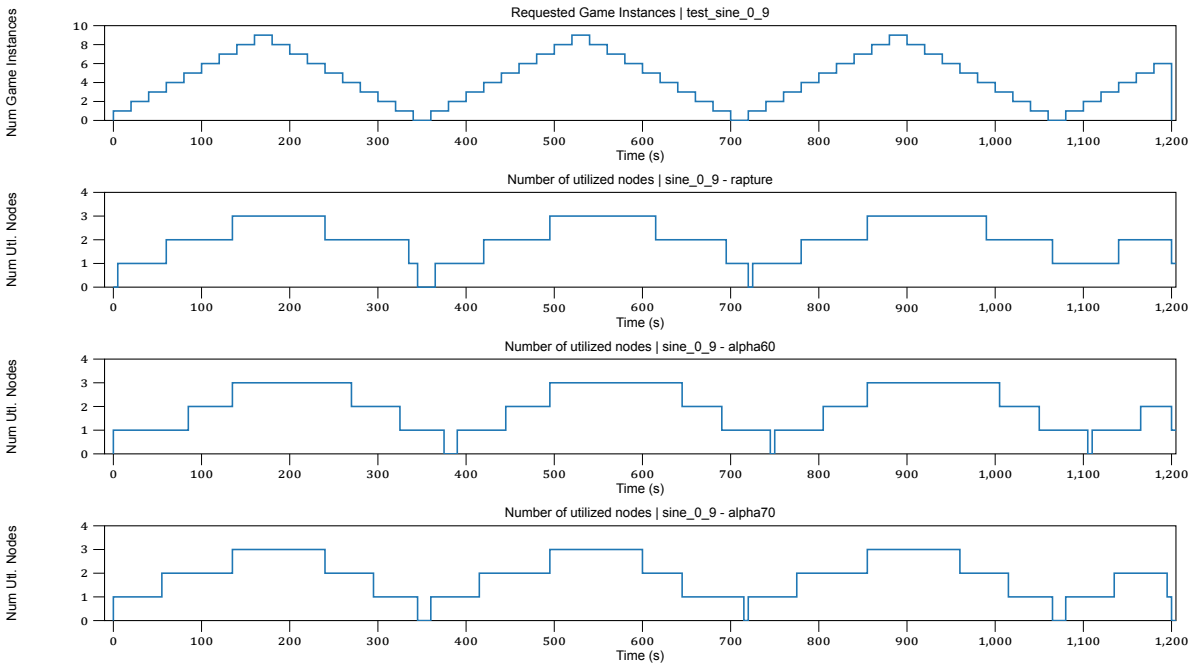


Figure 4.20: Node utilization for high load and sinusoidal incoming requests, comparing migration disabled and migration enabled for utilization thresholds of 60% and 70%.

The sinusoidal pattern was a more useful experiment to see how migration can better follow the decreasing demand. In Figure 4.20, a more aggressive migration behavior shows better node utilization. In Table 4.6 we see that for the sinusoidal test, migration has less of an improvement difference for both values of Alpha. This is because in a complete down-scaling situation, even the nodes that are low in utilization will be shut down in a short period of time and thus there is less room for optimization.

Table 4.6: Area under curve scores for node utilization

	Rapture	**Alpha 60%**	**Alpha 70%**
AUC score	2430	2380	2250
Improvement	32.5%	33.9%	37.5%

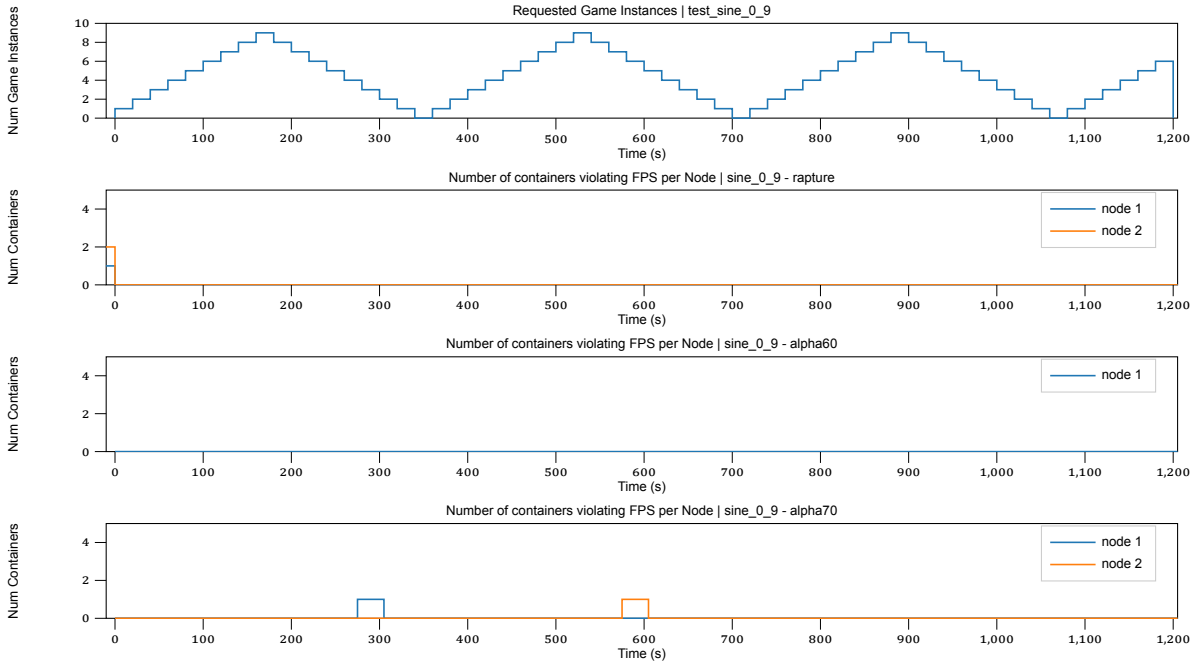


Figure 4.21: Frame rate violations for high load and sinusoidal incoming requests, comparing migration disabled and migration enabled for utilization thresholds of 60% and 70%

But in turn, from Figure 4.21 we noticed violations occurring for an alpha of 70%.

4.5. Conclusion

Concluding the experimentation chapter, we have elaborated the conducted experiments and their results. First, we have shown in section 4.1 that with software resource limitations we can create a stable and isolated multi-tenancy environment. Next, section 4.2 shows how the scheduler performs against regular scheduling strategies, given the metrics this scheduler receives and acts upon. Saving and restoring game instance state is briefly tested in 4.4. Finally, the node utilization benefits of migrating users from one physical machine to another has been demonstrated in 4.4.

5 Discussion

In this thesis we have introduced Rapture, an efficient cloud gaming platform. We introduced virtualization techniques to create a stable multi-tenant environment. The video game instances are scheduled efficiently compared to regular scheduling algorithms, while maintaining the desired quality of experience. A novel approach to Checkpoint and Restore by log and replay allows for live migration of video game instances. This migration allows for even a more tight-packed system in downscaling situations, with the cost of stability and loss of quality of experience. We now revisit the research questions from Chapter 1, and give answer by taking the findings from this thesis.

RQ1 What virtualization techniques can we implement to support multi-tenancy in a cloud gaming system?
Containers are a low-overhead and lightweight alternative to virtual machines. Since containers share the kernel with the host system, the GPU is available to the processes within the container. Containers allow for limiting CPU and Memory but not limiting GPU usage. We introduced frame rate and resolution limiting for the video game to limit the GPU's space and time usage. Results from experimentation show that frame rate and resolution lower GPU utilization in terms of video memory and bandwidth while also limiting CPU usage. Furthermore, experiments show that these resource restrictions provide a stable multi-tenant environment for a maximum number of video game instances to run concurrently on a single GPU. These tests were conducted with both a Linux and Windows build of the Viking Village benchmark, showing the capabilities of Rapture running video games built for different operating systems.

RQ2 How can we design a scheduler for cloud gaming systems?
Standard scheduling strategies do not consider GPU resources or schedule a container exclusively to a GPU. The current and legacy scheduling solutions only feature a reservation for CPU and memory. Rapture incorporates a monitoring and scheduling system built on Docker Swarm that checks both CPU and GPU utilization and includes a best-fit scheduling algorithm that looks at the space and time of GPU. In scheduling experiments, the Rapture scheduler shows better node utilization than a Spread strategy while not violating FPS requirements like a Binpack strategy does.

RQ3 How can we enable Checkpoint and Restore for Hardware-accelerated 3D Graphics?
Checkpointing hardware-accelerated applications are nontrivial. In this work, we demonstrated a solution that uses IPC via shared memory to decouple a graphical hardware-accelerated application into an upper and a lower half. For Checkpoint and Restore, the upper half is checkpointed by CRIU, and the lower half is logged and replayed. Serialization and synchronization were very important for this solution to work, but imposed overhead compared to normal execution. This was tested with Glxgears by hooking OpenGL calls. The result is that the graphical application can be preloaded with a shared library that decouples the executable into an upper and lower half and enables Checkpoint and Restore. Startup times have been shown to be reasonable.

RQ4 How can we facilitate efficient down-scaling by migration in a Cloud Gaming System?
With Checkpoint-Restart enabled, migration is possible. CRIU is used internally within the video

game container to checkpoint the upper and log-and-replay the lower half separately. A migration listener migrates containers when utilization is lower than a set threshold. The node with the lowest utilization is evacuated, and video game containers are rescheduled using the rapture scheduling algorithm. This was tested with Viking Village without a Checkpoint and Restore solution, for migration thresholds 60% and 70%, with 70% being more aggressive. We see that a more aggressive migration strategy allows for a more tightly packed system over time, but when one incurs a cost of migration, this might not be the optimal strategy.

5.1. Limitations

The research of Rapture is not an all encompassing work. We state the following limitations:

- This research has elaborated on the many benefits of containers versus virtual machines. The performance of containers has been tested with the Viking Village benchmark in Chapter 4 and shows promising results. However, no direct comparison with the Viking Village benchmark on a virtual machine has been made. Multiple attempts have been made to set up a machine running a hypervisor with GPU passthrough. However, the tedious setup within a cloud service provider such as Google Cloud resulted in many failed attempts.

- The scope of this research has been limited to creating a platform that can schedule and run video game instances. This means that a big aspect of a cloud gaming service was left out, namely the streaming. While early attempts have been made to set up a streaming client with WebRTC, initial results showed poor performance with complex implementations, and we redirected focus to platform solutions.

  Video encoding can be hardware accelerated and is demanding to the GPU. It would be interesting to see the system performance with this video compression and streaming client in place. Furthermore, an interactive user test could verify the actual viability of the cloud gaming system. It could also give an interesting insight into how a user experiences migration sequences.

- We took only two applications for benchmarking and testing. While for Viking Village, we tested both a Linux and Windows build, however, this benchmark does not switch model scenes. When changing a model scene, the VRAM gets repopulated with new geometries and textures and could test multi-tenancy stability. However, the system has proven to be stable under game instances populating and leaving, which also involves context creation and destruction.

  Finally, we only tested the scheduling and migration with a single type of application, namely the Linux build of Viking Village. Applications with different resource demands could give interesting results while being scheduled and run Rapture.

- We developed the solution of enabling Checkpoint and Restore for GPUs with IPC via shared memory with a minimal example. Next to the fact that solutions posed within research (Section 2.5) show better performance, the implementation of IPC via shared memory solution was done with a simple application such as Glxgears and a limited set of functions. Glxgears was also slightly adapted, which does away with the “transparency.”

  Glxgears also creates a persistent graphical state within the first thousand calls, at which there is no object and scene creation or destruction for the rest of the execution. This makes it a weak validator for the log and replay approach that is used in the IPC via shared memory solution. The log is also pruned naively by simply removing a set of frames.

  Finally, the tests with Viking Village have been done with Vulkan, and the C/R solution is made for OpenGL. A solution for Vulkan would have tied everything together but was not pursued, as OpenGL is less complex for small applications.

- The IPC via shared memory solution was only tested locally, and could not be fully implemented in the migration experiments. This experiment would have brought every aspect of this thesis together. Unfortunately, while everything is implemented in the Rapture scheduler, Docker Swarm misses the capability of running privileged containers as root. An implementation on Kubernetes could work as it does have options to run privileged containers in pods and would make sending checkpoint commands easier via the ”pod exec” command.

5.2. Recommendations for Future Work

In this section we denote a few findings from this research that pose an interesting direction of research in the future.

- The idea of containerizing video games and imposing resource restrictions via frame rate and resolution limiting is a universal solution regarding operating systems. The solution we posed for running Microsoft Windows developed video games was introducing a compatibility layer called Wine, with the DXVK extension to translate the Direct3D calls to Vulkan. This method imposes significant overhead, and thus we should research other solutions. Containerization technologies or process isolation frameworks on Windows Server could have the plethora of Windows native games run in resource-restricted environments, with no overhead of translation layers. These technologies are less mature than containerization technologies on Linux; a cloud gaming platform on Windows Server is nevertheless an exciting research topic.

- Much research is focused on passthrough, multiplexing, and multi-tenancy of GPUs within GPGPU computing but not so much on graphical computing and cloud gaming. As this market is bound to grow and graphical offloading to the cloud becomes commonplace when bandwidth reaches sufficient capacity, it is worthwhile to apply more research for both containers and virtual machines in this direction. GPU passthrough has been shown to improve over the years with updates to Hypervisors such as Citrix XenServer and KVM. Benchmarking performance for graphical offloading to the cloud contributes to an up-to-date understanding of the capabilities of cloud graphics processing.

- Transparent Checkpoint and Restore technologies, such as CRIU, are exciting tools for cloud computing. Migration enabled by C/R can help in mobile edge-computing situations, infrastructure reorganization, and checkpointing and restoring fault-tolerant applications improve stability.

  GPU vendors, like NVIDIA, have recently started to open-source their GPU drivers. This gives way to further research into extending C/R technologies for hardware-accelerated workloads. Plugins that handle device memory maps can be developed for CRIU, and driver open-sourcing enables this.

  Nevertheless, considering the techniques we have used in this research, logging and replaying the lower half of the split process to enable C/R requires carefully monitoring calls made to APIs interfacing with the GPUs. In work that has investigated GPU sharing, most of the solutions try to monitor and limit the calls made to the GPU in a similar way. This creates a compelling research area that controls GPUs on a library level and that could have more applications within hardware-accelerated machine learning of video gaming.

Bibliography

[1] Andrew S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[2] Peter Eisert and Philipp Fechteler. Remote rendering of computer games. *SIGMAP 2007 - International Conference on Signal Processing and Multimedia Applications, Proceedings*, (May):438–443, 2007.

[3] Chun-Ying Huang, Cheng-Hsin Hsu, and Kuan-Ta Chen. GamingAnywhere. *ACM SIGMultimedia Records*, 7(1):3–5, 2015.

[4] Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. *IEEE/ACM Transactions on Networking*, 24(4):2128–2139, 2016.

[5] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software*, pages 171–172, 2015.

[6] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 386–393, 2015.

[7] Amit M. Potdar, D. G. Narayan, Shivaraj Kengond, and Mohammed Moin Mulla. Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science*, 171(2019):1419–1428, 2020.

[8] Micah Dowty and Jeremy Sugerman. GPU virtualization on VMware's hosted I/O architecture. *Operating Systems Review (ACM)*, 43(3):73–82, 2009.

[9] Franck Diard. Cloud Gaming & Application Delivery with NVIDIA GRID Technologies. *GPU Technology Conference*.

[10] AMD. GPU Virtualization Solution | MxGPU Technology | AMD, 2020.

[11] Andrew J. Younge, John Paul Walters, Stephen Crago, and Geoffrey C. Fox. Evaluating GPU passthrough in xen for high performance cloud computing. *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, pages 852–859, 2014.

[12] John Paul Walters, Andrew J. Younge, Dong In Kang, Ke Thia Yao, Mikyung Kang, Stephen P. Crago, and Geoffrey C. Fox. GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and openCL applications. *IEEE International Conference on Cloud Computing, CLOUD*, pages 636–643, 2014.

[13] Ryan Shea and Jiangchuan Liu. On GPU pass-through performance for cloud gaming: Experiments and analysis. *Annual Workshop on Network and Systems Support for Games*, pages 1–6, 2013.

[14] Qingdong Hou, Chu Qiu, Kaihui Mu, Quan Qi, and Yongquan Lu. A cloud gaming system based on NVIDIA GRID GPU. *Proceedings - 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, DCABES 2014*, pages 73–77, 2014.

[15] Teemu Kamarainen, Yuanqi Shan, Matti Siekkinen, and Antti Yla-Jaaski. Virtual machines vs. containers in cloud gaming systems. *Annual Workshop on Network and Systems Support for Games*, 2016-Janua, 2016.

[16] Daeyoun Kang, Tae Joon Jun, Dohyeun Kim, Jaewook Kim, and Daeyoung Kim. ConVGPU: GPU Management Middleware in Container Based Virtualized Environment. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2017-Septe:301–309, 2017.

[17] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. GaiaGPU: Sharing GPUs in container clouds. *Proceedings - 16th IEEE International Symposium on Parallel and Distributed Processing with Applications, 17th IEEE International Conference on Ubiquitous Computing and Communications, 8th IEEE International Conference on Big Data and Cloud Computing, 11t*, pages 469–476, 2019.

[18] Shaleen Garg, Kishore Kothapalli, and Suresh Purini. Share-a-GPU: Providing Simple and Effective Time-Sharing on GPUs. *Proceedings - 25th IEEE International Conference on High Performance Computing, HiPC 2018*, pages 294–303, 2019.

[19] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-Gaming: A Cost-Efficient Cloud Gaming System at Scale. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2849–2865, 2019.

[20] Bob Amstadt and Michael K. Johnson. Wine. *Linux Journal*, 1994:3–es, 1994.

[21] Wei Zhang, Xiaofei Liao, Peng Li, Hai Jin, and Li Lin. ShareRender: Bypassing GPU virtualization to enable fine-grained resource sharing for cloud gaming. *MM 2017 - Proceedings of the 2017 ACM Multimedia Conference*, pages 324–332, 2017.

[22] Shangguang Wang, Jinliang Xu, Ning Zhang, and Yujiong Liu. A Survey on Service Migration in Mobile Edge Computing. *IEEE Access*, 6:23511–23528, 2018.

[23] Andrey Mirkin. Containers checkpointing and live migration. *Annals of Oncology*, 22:iv5, 2011.

[24] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494–499, 2006.

[25] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. *IPDPS 2009 - Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009.

[26] Checkpoint/Restore In Userspace (CRIU), 2012.

[27] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2019-Septe, 2019.

[28] Nishant Deepak Keni and Ahan Kak. Adaptive Containerization for Microservices in Distributed Cloud Systems. *2020 IEEE 17th Annual Consumer Communications and Networking Conference, CCNC 2020*, 2020.

[29] Felix Kuehling, Rajneesh Bhardwaj, and David Yat Sin. Fast Checkpoint Restore for AMD GPUs with CRIU. pages 1–13, 2021.

[30] Rohan Garg, Gregory Price, and Gene Cooperman. MANA for MPI: MPI-Agnostic Network-Agnostic transparent checkpointing. *HPDC 2019- Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 49–60, 2019.

[31] Twinkle Jain and Gene Cooperman. Crac: Checkpoint-restart architecture for cuda with streams and uvm. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2020-Novem, 2020.

[32] Samaneh Kazemi Nafchi, Rohan Garg, and Gene Cooperman. Transparent Checkpoint-Restart for Hardware-Accelerated 3D Graphics. pages 1–20, 2013.

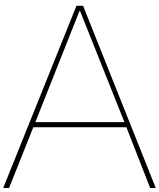
[33] David Hou, Jun Gan, Yue Li, Younes El Idrissi Yazami, and Twinkle Jain. Transparent Checkpointing for OpenGL Applications on GPUs. pages 4–6, 2021.

[34] Atsushi Hori, Masamichi Takagi, Min Si, Jai Dayal, Yutaka Ishikawa, Balazs Gerofi, and Pavan Balaji. Process-in-process: Techniques for practical address-space sharing. *HPDC 2018 - Proceedings of the 2018 International Symposium on High-Performance Parallel and Distributed Computing*, pages 131–143, 2018.

[35] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. *IEEE International Symposium on High Performance Distributed Computing, Proceedings*, (January):177–184, 1999.

[36] Tristan Braud, Ahmad Alhilal, and Pan Hui. Talaria. pages 375–381, 2021.

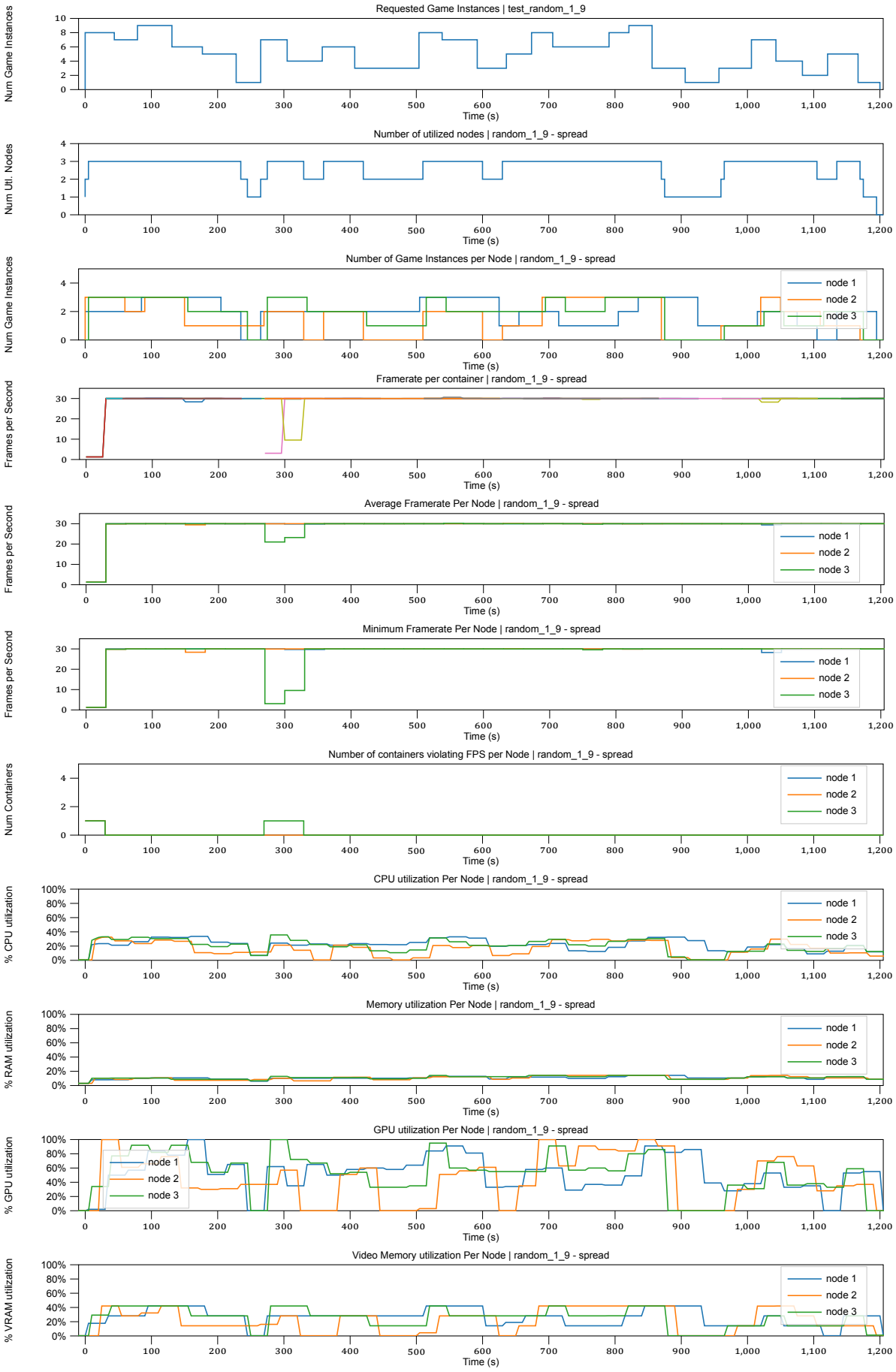
[37] Yunbiao Lin, Wei Wang, and Kyle Gui. OpenGL application live migration with GPU acceleration in personal cloud. *HPDC 2010 - Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 280–283, 2010.



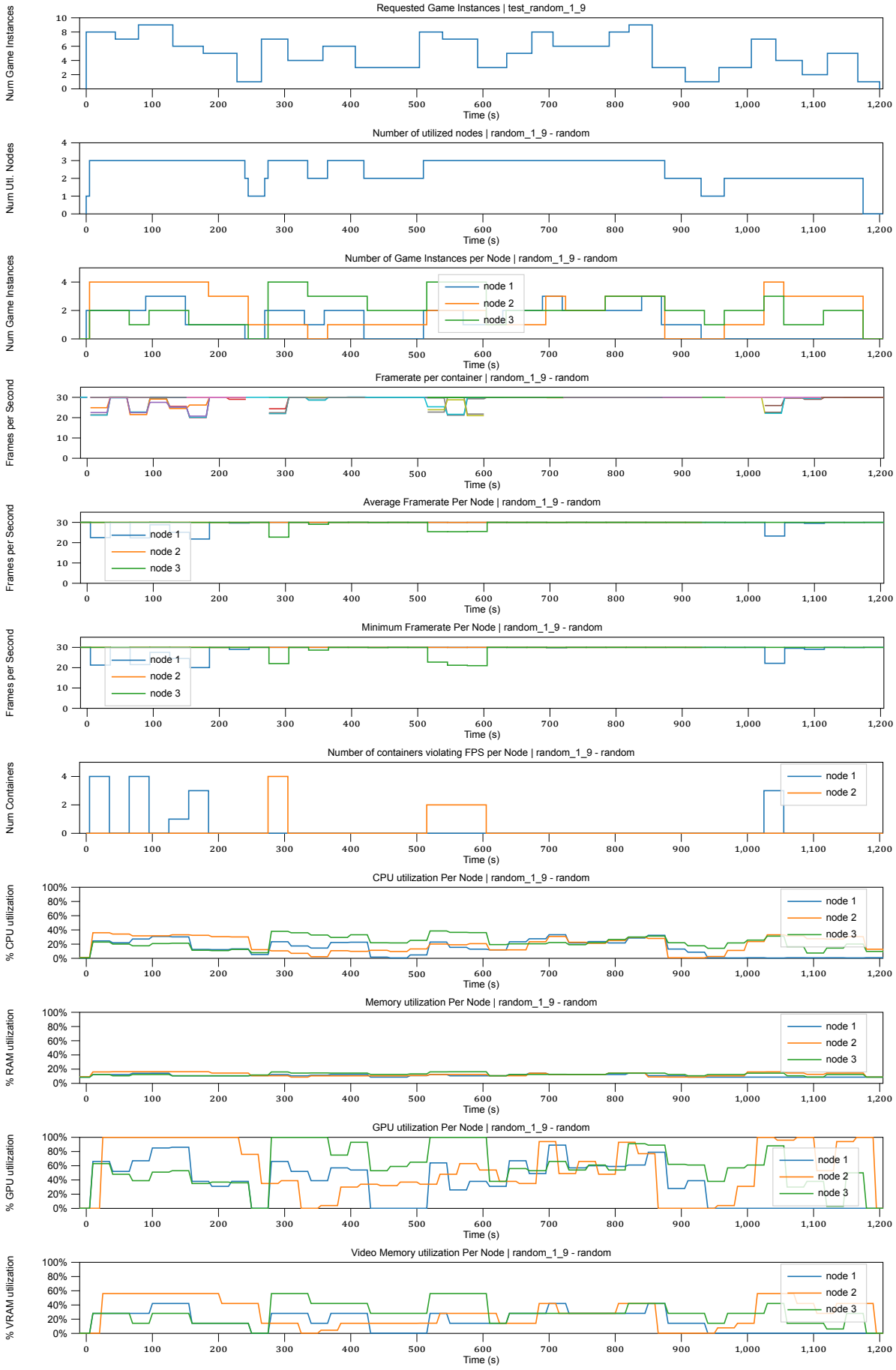
Scheduler Experiments

A.1. High Load Random Requests

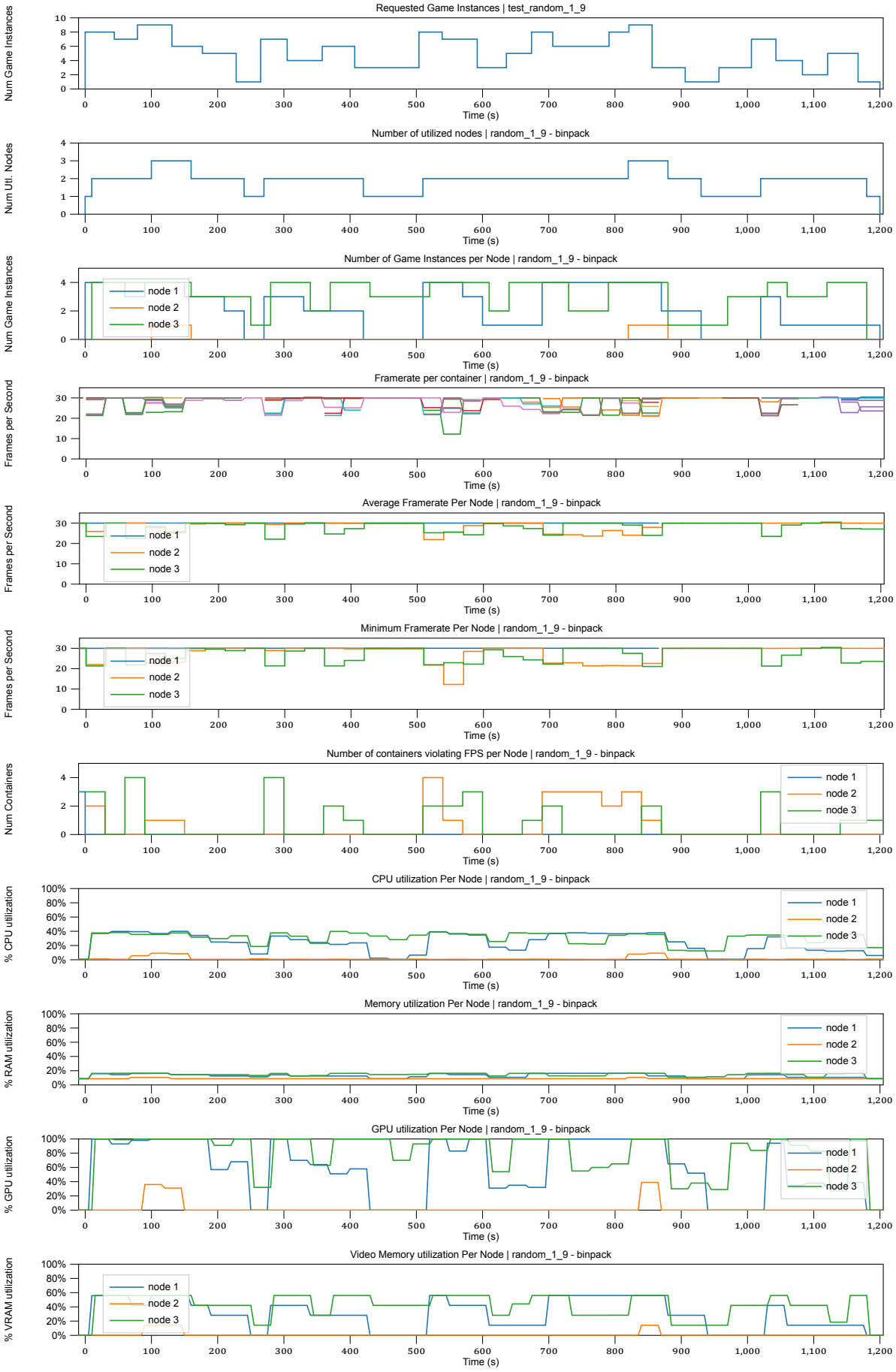
Spread



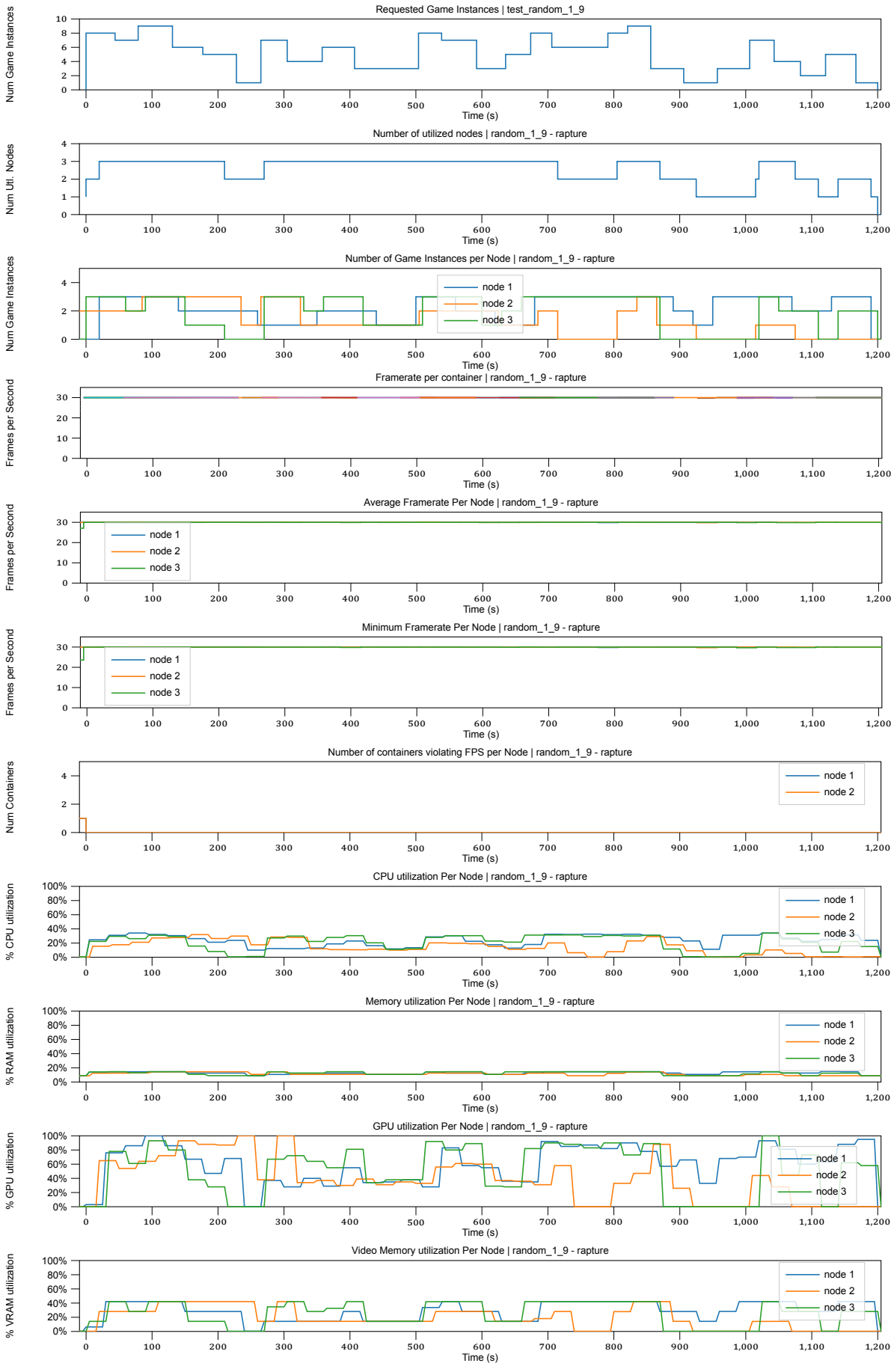
Random


Requested Game Instances | test_random_1_9
Num Game Instances
Time (s)
Number of utilized nodes | random_1_9 - random
Num Util. Nodes
Number of Game Instances per Node | random_1_9 - random
node 1
node 2
node 3
Framerate per container | random_1_9 - random
Frames per Second
Average Framerate Per Node | random_1_9 - random
Minimum Framerate Per Node | random_1_9 - random
Number of containers violating FPS per Node | random_1_9 - random
Num Containers
CPU utilization Per Node | random_1_9 - random
% CPU utilization
Memory utilization Per Node | random_1_9 - random
% RAM utilization
GPU utilization Per Node | random_1_9 - random
% GPU utilization
Video Memory utilization Per Node | random_1_9 - random
% VRAM utilization

Binpack

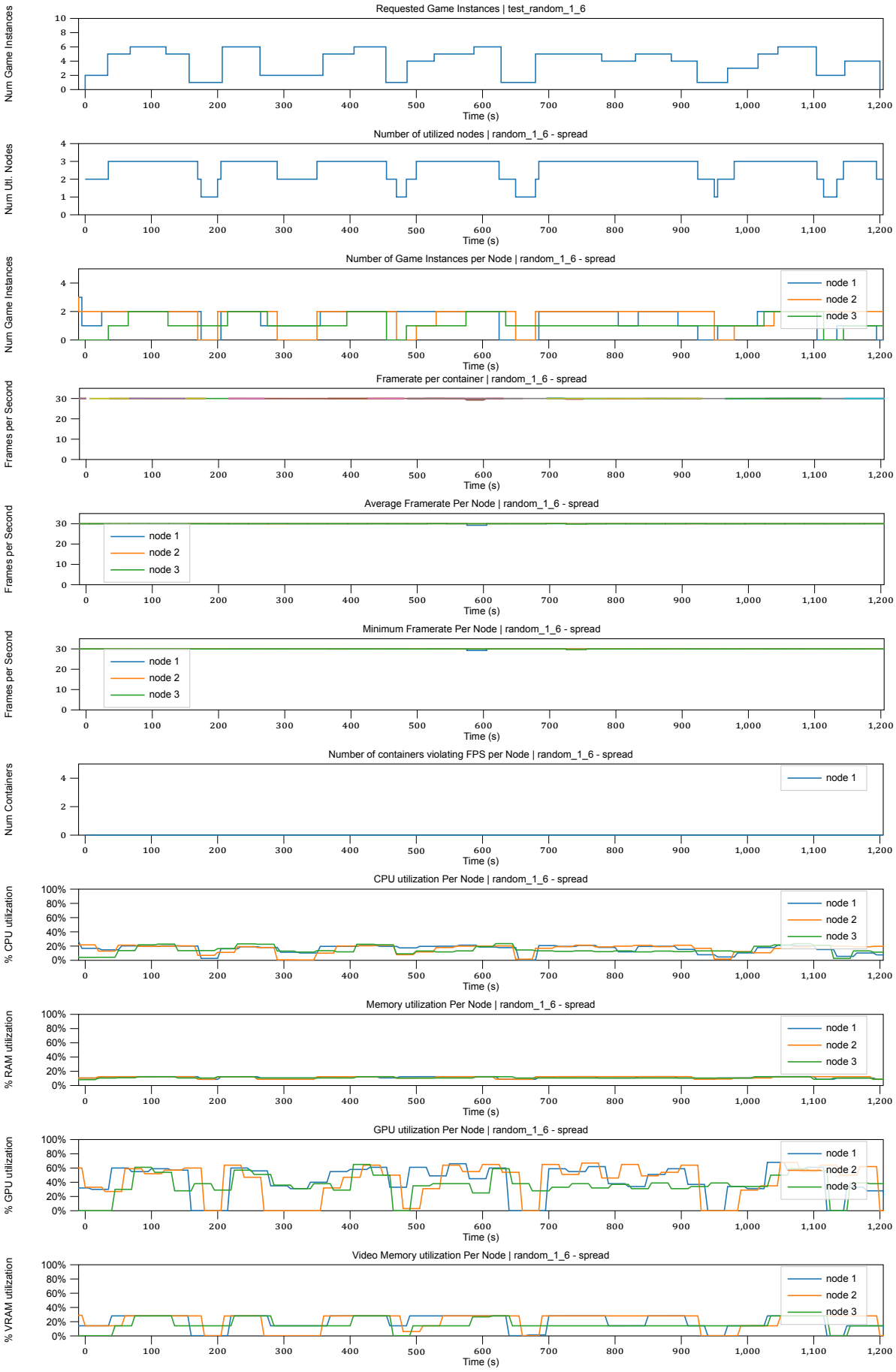

Requested Game Instances | test_random_1_9
Num Game Instances
Time (s)
Number of utilized nodes | random_1_9 - binpack
Num Util. Nodes
Number of Game Instances per Node | random_1_9 - binpack
node 1
node 2
node 3
Framerate per container | random_1_9 - binpack
Frames per Second
Average Framerate Per Node | random_1_9 - binpack
Minimum Framerate Per Node | random_1_9 - binpack
Number of containers violating FPS per Node | random_1_9 - binpack
Num Containers
CPU utilization Per Node | random_1_9 - binpack
% CPU utilization
Memory utilization Per Node | random_1_9 - binpack
% RAM utilization
GPU utilization Per Node | random_1_9 - binpack
% GPU utilization
Video Memory utilization Per Node | random_1_9 - binpack
% VRAM utilization

Rapture

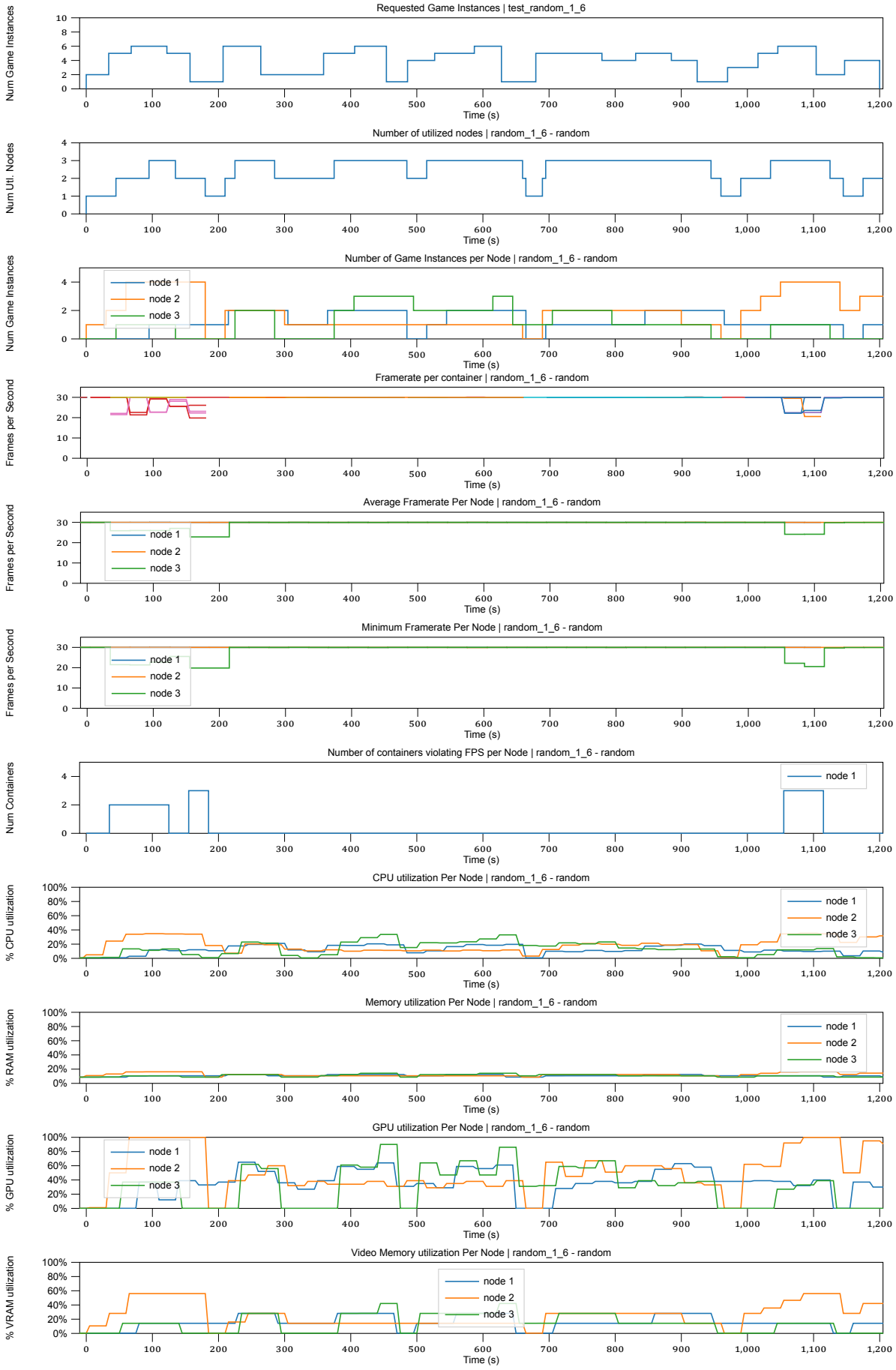

Requested Game Instances | test_random_1_9
Num Game Instances
Time (s)
Number of utilized nodes | random_1_9 - rapture
Num Util. Nodes
Number of Game Instances per Node | random_1_9 - rapture
node 1
node 2
node 3
Framerate per container | random_1_9 - rapture
Frames per Second
Average Framerate Per Node | random_1_9 - rapture
Minimum Framerate Per Node | random_1_9 - rapture
Number of containers violating FPS per Node | random_1_9 - rapture
Num Containers
CPU utilization Per Node | random_1_9 - rapture
% CPU utilization
Memory utilization Per Node | random_1_9 - rapture
% RAM utilization
GPU utilization Per Node | random_1_9 - rapture
% GPU utilization
Video Memory utilization Per Node | random_1_9 - rapture
% VRAM utilization

A.2. Low Load Random Requests

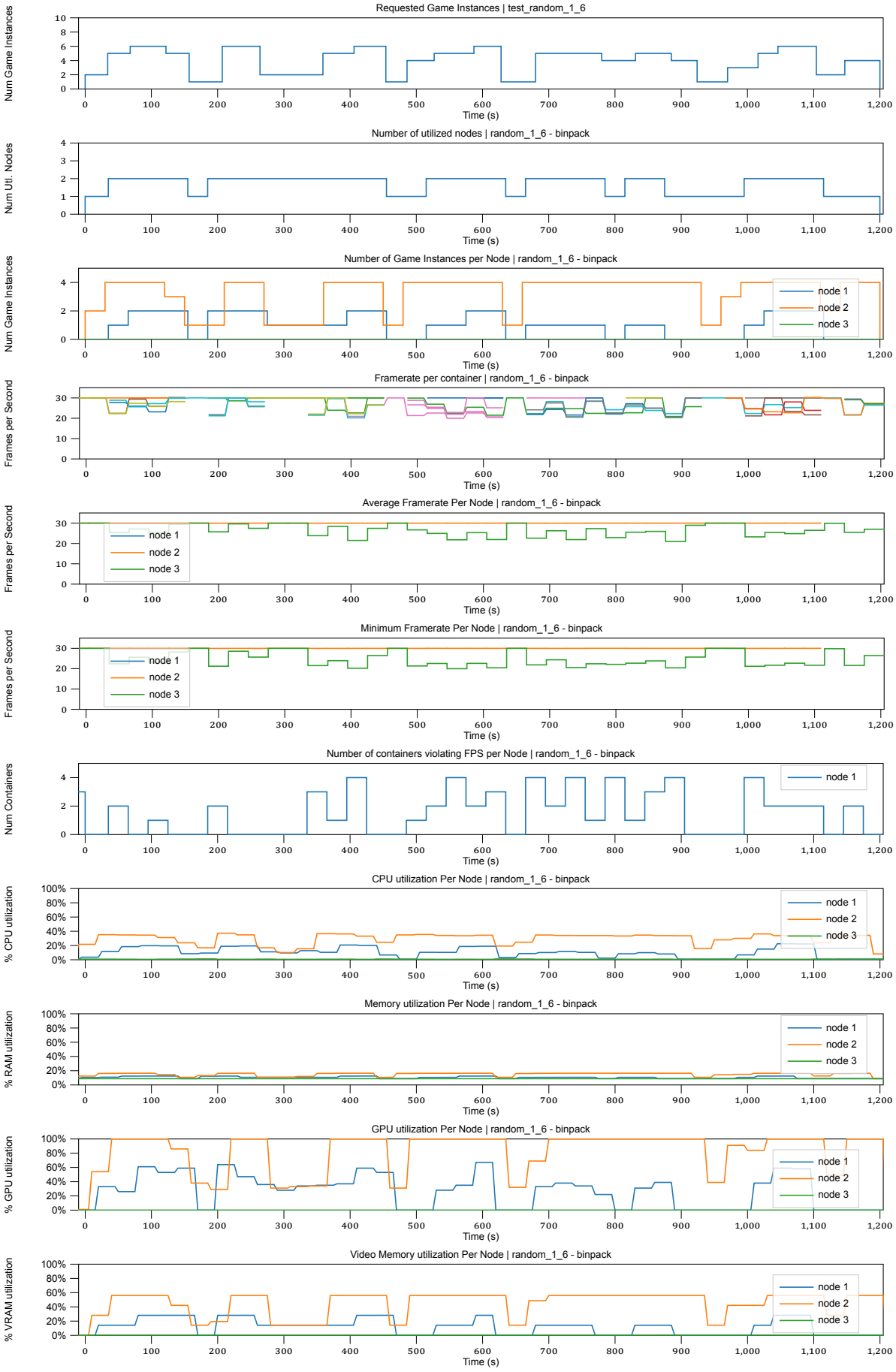
Spread



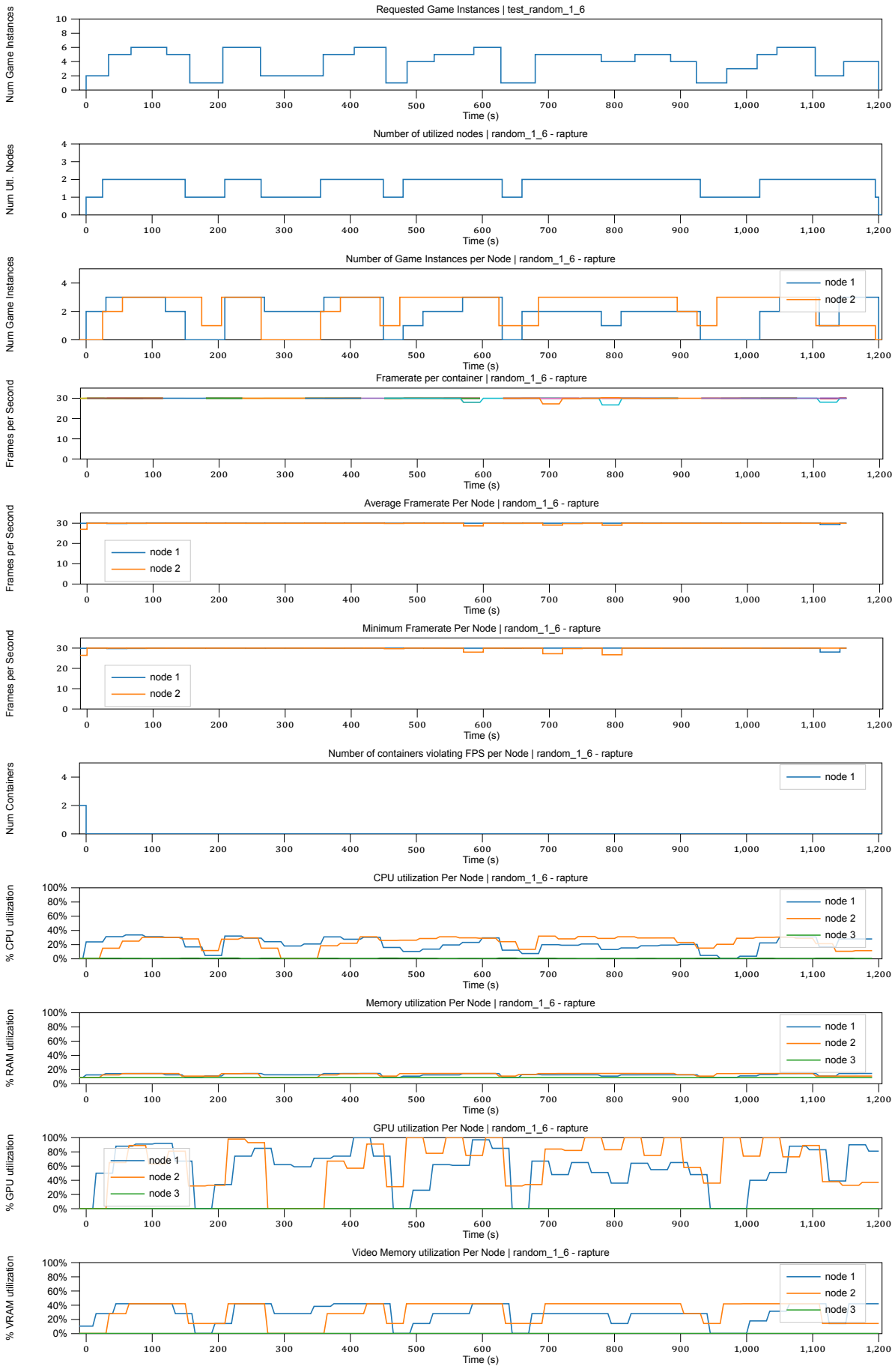
Random


Requested Game Instances | test_random_1_6
Num Game Instances
Time (s)
Number of utilized nodes | random_1_6 - random
Num Util. Nodes
Number of Game Instances per Node | random_1_6 - random
node 1
node 2
node 3
Framerate per container | random_1_6 - random
Frames per Second
Average Framerate Per Node | random_1_6 - random
Minimum Framerate Per Node | random_1_6 - random
Number of containers violating FPS per Node | random_1_6 - random
Num Containers
CPU utilization Per Node | random_1_6 - random
% CPU utilization
Memory utilization Per Node | random_1_6 - random
% RAM utilization
GPU utilization Per Node | random_1_6 - random
% GPU utilization
Video Memory utilization Per Node | random_1_6 - random
% VRAM utilization

Binpack

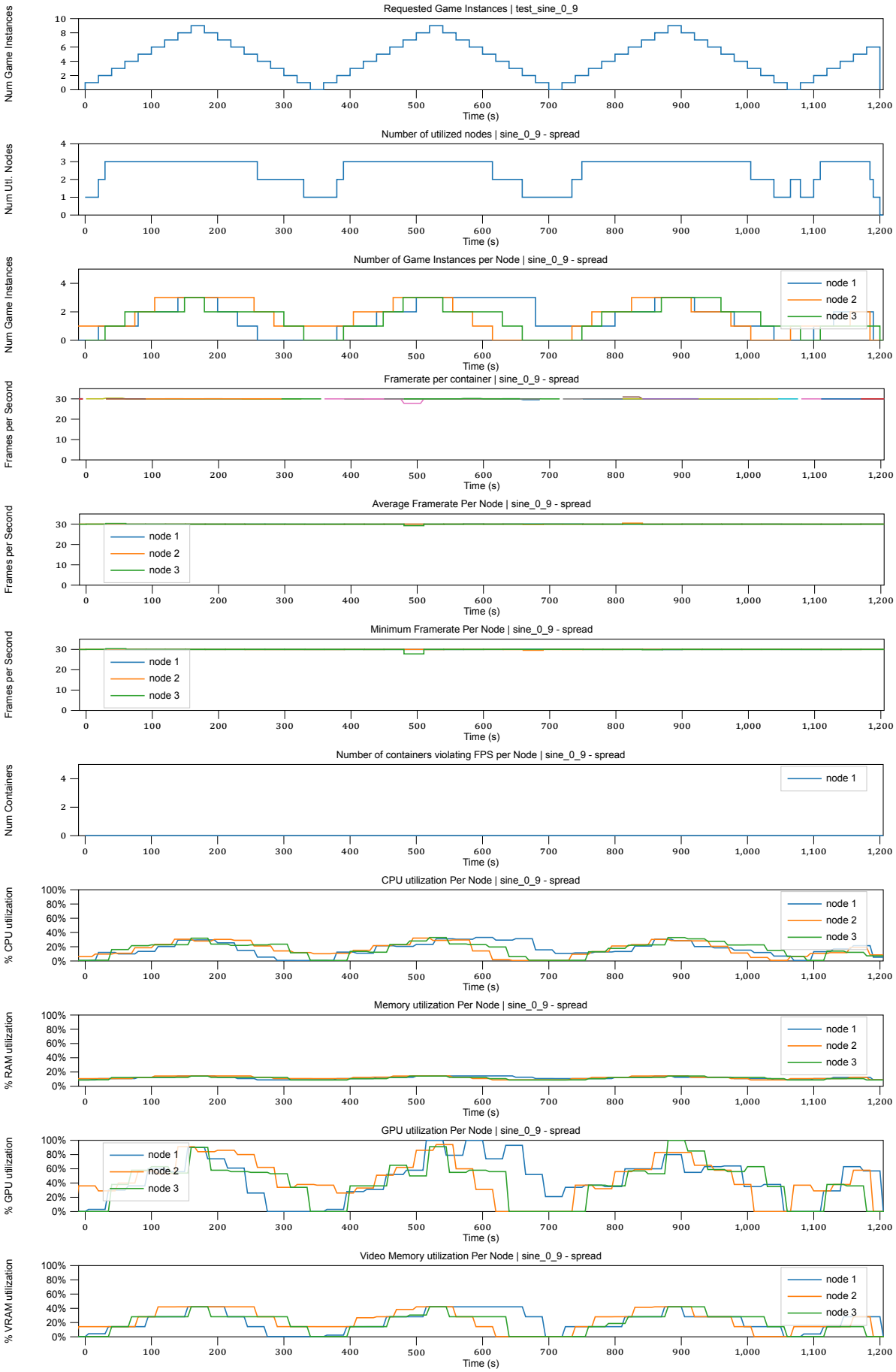

Requested Game Instances | test_random_1_6
Num Game Instances
Time (s)
Number of utilized nodes | random_1_6 - binpack
Num Util. Nodes
Number of Game Instances per Node | random_1_6 - binpack
node 1
node 2
node 3
Framerate per container | random_1_6 - binpack
Frames per Second
Average Framerate Per Node | random_1_6 - binpack
Minimum Framerate Per Node | random_1_6 - binpack
Number of containers violating FPS per Node | random_1_6 - binpack
Num Containers
CPU utilization Per Node | random_1_6 - binpack
% CPU utilization
Memory utilization Per Node | random_1_6 - binpack
% RAM utilization
GPU utilization Per Node | random_1_6 - binpack
% GPU utilization
Video Memory utilization Per Node | random_1_6 - binpack
% VRAM utilization

Rapture

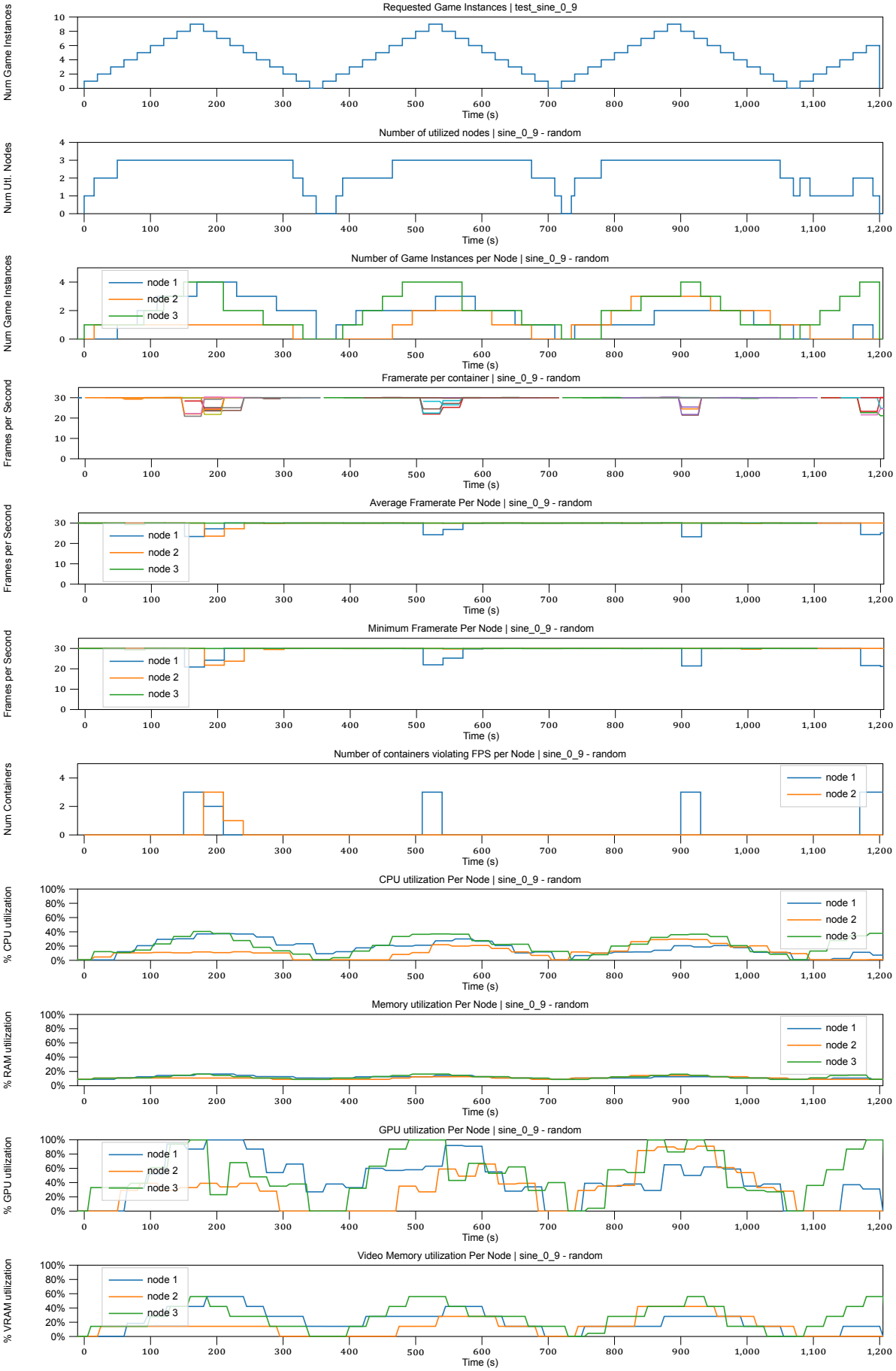

Requested Game Instances | test_random_1_6
Num Game Instances
Time (s)
Number of utilized nodes | random_1_6 - rapture
Num Util. Nodes
Number of Game Instances per Node | random_1_6 - rapture
node 1
node 2
Framerate per container | random_1_6 - rapture
Frames per Second
Average Framerate Per Node | random_1_6 - rapture
Minimum Framerate Per Node | random_1_6 - rapture
Number of containers violating FPS per Node | random_1_6 - rapture
Num Containers
CPU utilization Per Node | random_1_6 - rapture
% CPU utilization
node 3
Memory utilization Per Node | random_1_6 - rapture
% RAM utilization
GPU utilization Per Node | random_1_6 - rapture
% GPU utilization
Video Memory utilization Per Node | random_1_6 - rapture
% VRAM utilization

A.3. High Load Sine Requests

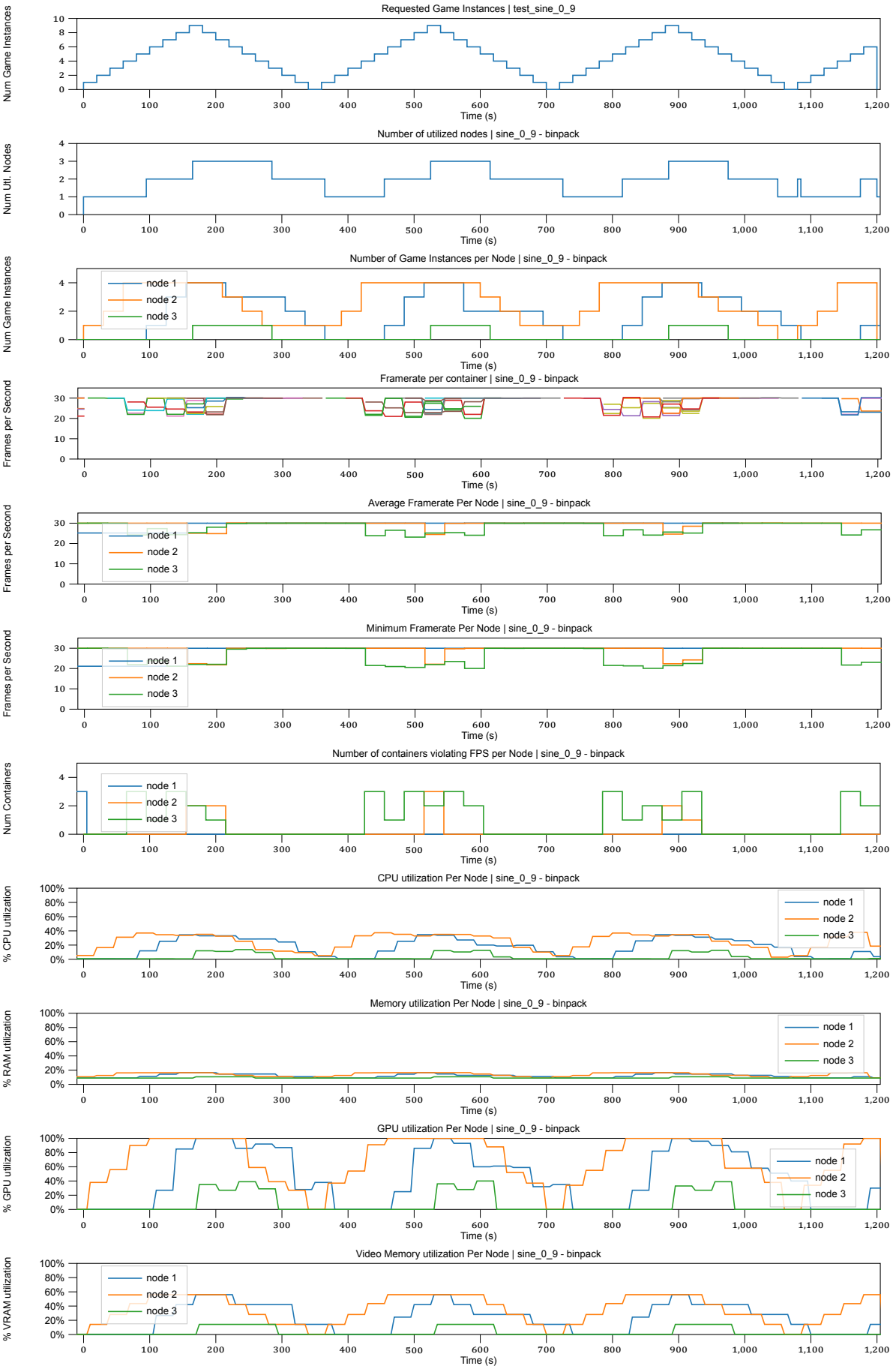
Spread



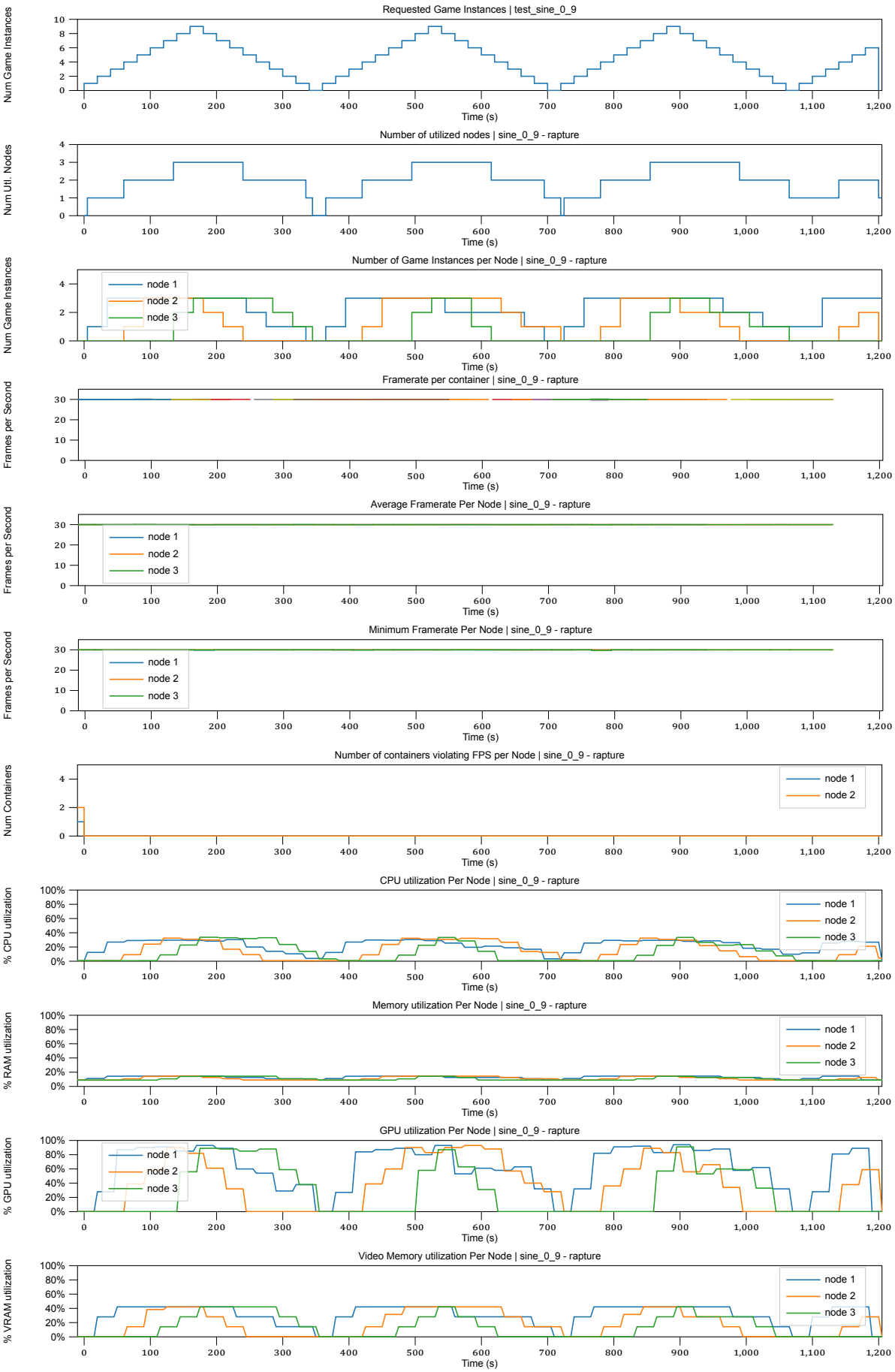
Random


Requested Game Instances | test_sine_0_9
Num Game Instances
Time (s)
Number of utilized nodes | sine_0_9 - random
Num Util. Nodes
Number of Game Instances per Node | sine_0_9 - random
node 1
node 2
node 3
Framerate per container | sine_0_9 - random
Frames per Second
Average Framerate Per Node | sine_0_9 - random
Minimum Framerate Per Node | sine_0_9 - random
Number of containers violating FPS per Node | sine_0_9 - random
Num Containers
CPU utilization Per Node | sine_0_9 - random
% CPU utilization
Memory utilization Per Node | sine_0_9 - random
% RAM utilization
GPU utilization Per Node | sine_0_9 - random
% GPU utilization
Video Memory utilization Per Node | sine_0_9 - random
% VRAM utilization

Binpack

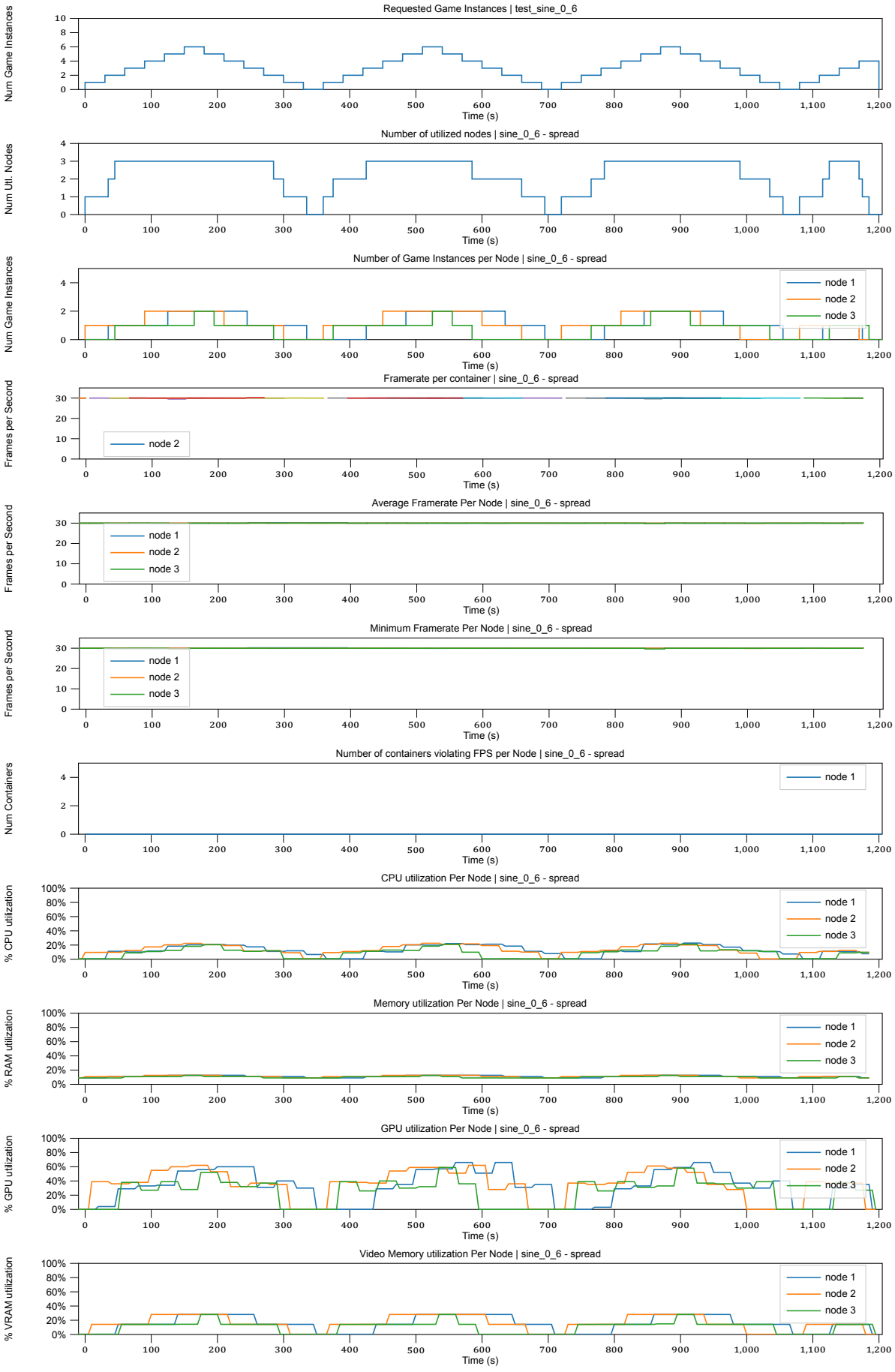

Requested Game Instances | test_sine_0_9
Num Game Instances
Time (s)
Number of utilized nodes | sine_0_9 - binpack
Num Uti. Nodes
Number of Game Instances per Node | sine_0_9 - binpack
node 1
node 2
node 3
Framerate per container | sine_0_9 - binpack
Frames per Second
Average Framerate Per Node | sine_0_9 - binpack
Minimum Framerate Per Node | sine_0_9 - binpack
Number of containers violating FPS per Node | sine_0_9 - binpack
Num Containers
CPU utilization Per Node | sine_0_9 - binpack
% CPU utilization
Memory utilization Per Node | sine_0_9 - binpack
% RAM utilization
GPU utilization Per Node | sine_0_9 - binpack
% GPU utilization
Video Memory utilization Per Node | sine_0_9 - binpack
% VRAM utilization

Rapture

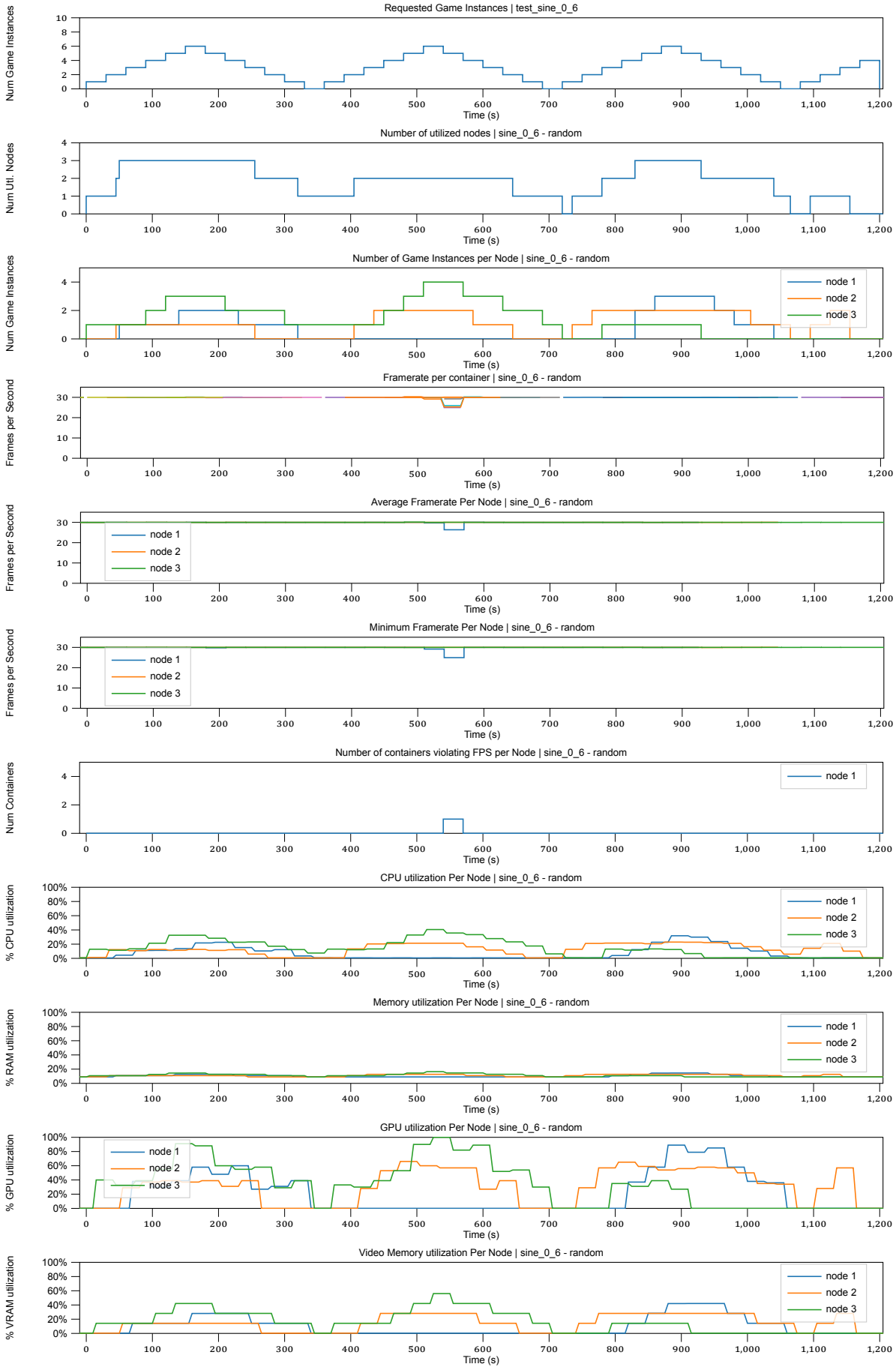

Requested Game Instances | test_sine_0_9
Num Game Instances
Time (s)
Number of utilized nodes | sine_0_9 - rapture
Num Util. Nodes
Number of Game Instances per Node | sine_0_9 - rapture
node 1
node 2
node 3
Framerate per container | sine_0_9 - rapture
Frames per Second
Average Framerate Per Node | sine_0_9 - rapture
Minimum Framerate Per Node | sine_0_9 - rapture
Number of containers violating FPS per Node | sine_0_9 - rapture
Num Containers
CPU utilization Per Node | sine_0_9 - rapture
% CPU utilization
Memory utilization Per Node | sine_0_9 - rapture
% RAM utilization
GPU utilization Per Node | sine_0_9 - rapture
% GPU utilization
Video Memory utilization Per Node | sine_0_9 - rapture
% VRAM utilization

A.4. Low Load Sine Requests

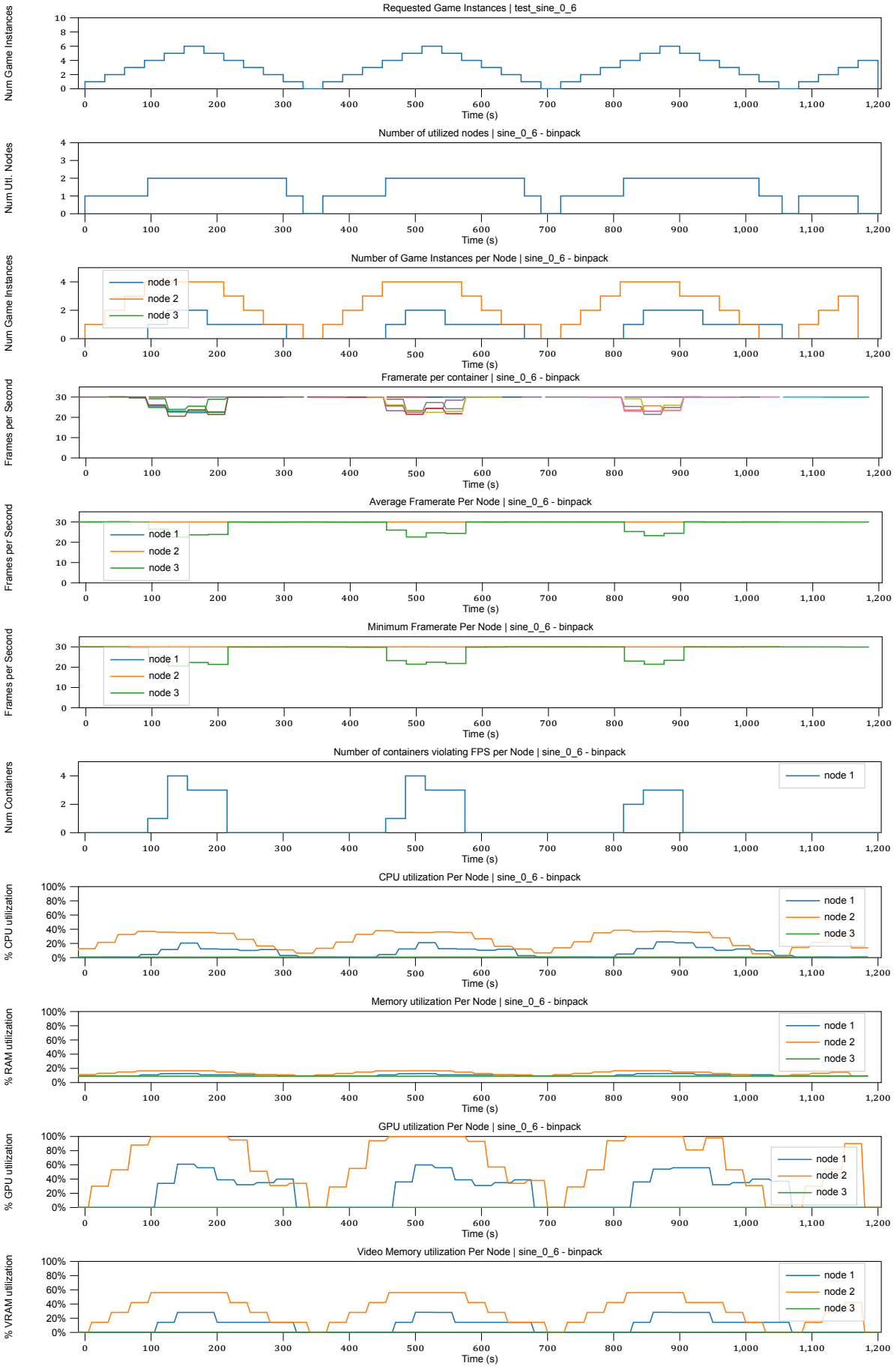
Spread



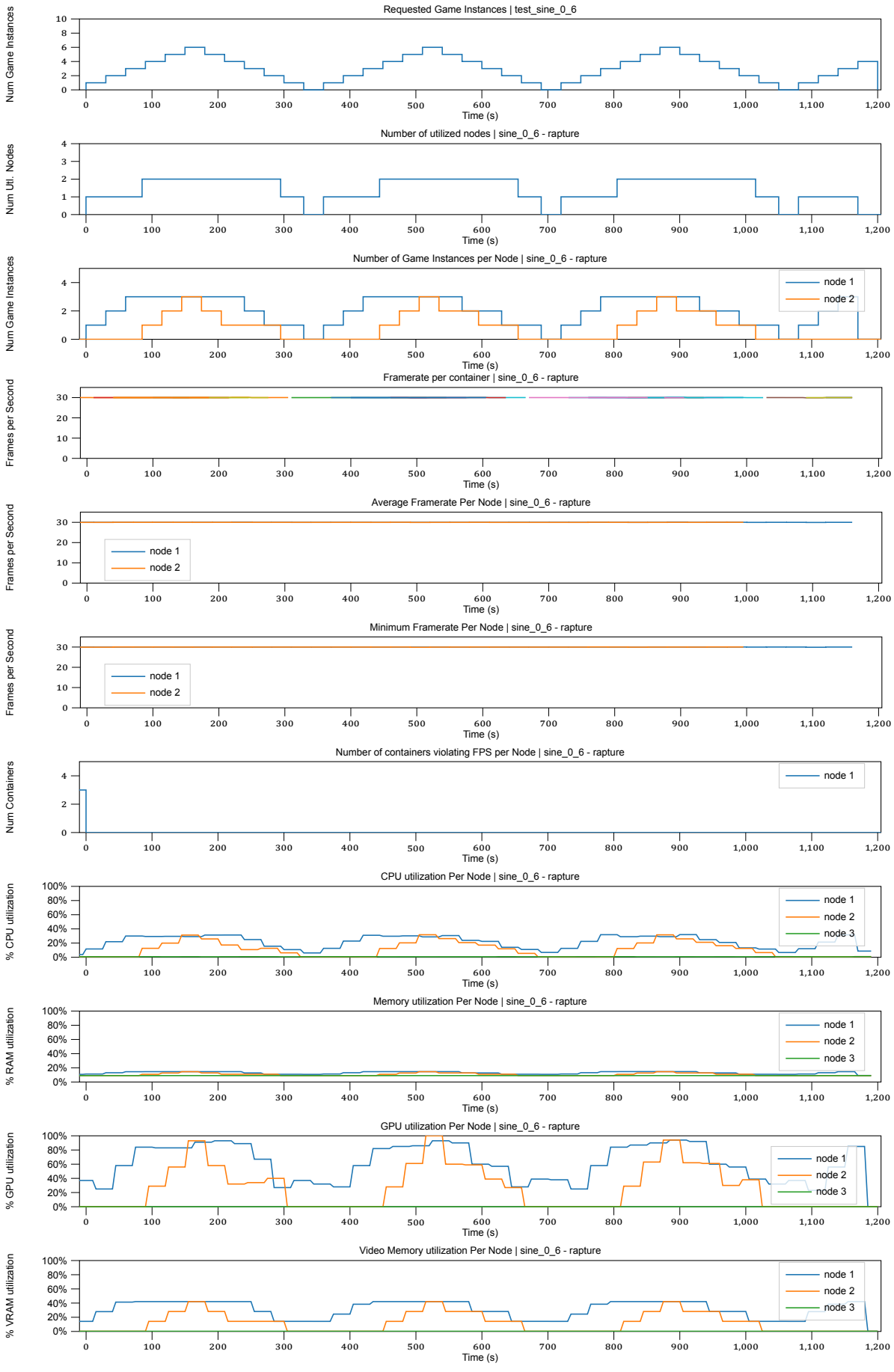
Random


Requested Game Instances | test_sine_0_6
Num Game Instances
Time (s)
Number of utilized nodes | sine_0_6 - random
Num Util. Nodes
Number of Game Instances per Node | sine_0_6 - random
node 1
node 2
node 3
Framerate per container | sine_0_6 - random
Frames per Second
Average Framerate Per Node | sine_0_6 - random
Minimum Framerate Per Node | sine_0_6 - random
Number of containers violating FPS per Node | sine_0_6 - random
Num Containers
CPU utilization Per Node | sine_0_6 - random
% CPU utilization
Memory utilization Per Node | sine_0_6 - random
% RAM utilization
GPU utilization Per Node | sine_0_6 - random
% GPU utilization
Video Memory utilization Per Node | sine_0_6 - random
% VRAM utilization

Binpack


Requested Game Instances | test_sine_0_6
Num Game Instances
Time (s)
Number of utilized nodes | sine_0_6 - binpack
Num Util. Nodes
Number of Game Instances per Node | sine_0_6 - binpack
node 1
node 2
node 3
Framerate per container | sine_0_6 - binpack
Frames per Second
Average Framerate Per Node | sine_0_6 - binpack
Minimum Framerate Per Node | sine_0_6 - binpack
Number of containers violating FPS per Node | sine_0_6 - binpack
Num Containers
CPU utilization Per Node | sine_0_6 - binpack
% CPU utilization
Memory utilization Per Node | sine_0_6 - binpack
% RAM utilization
GPU utilization Per Node | sine_0_6 - binpack
% GPU utilization
Video Memory utilization Per Node | sine_0_6 - binpack
% VRAM utilization

Rapture


Requested Game Instances | test_sine_0_6
Num Game Instances
Time (s)
Number of utilized nodes | sine_0_6 - rapture
Num Util. Nodes
Number of Game Instances per Node | sine_0_6 - rapture
node 1
node 2
Framerate per container | sine_0_6 - rapture
Frames per Second
Average Framerate Per Node | sine_0_6 - rapture
Minimum Framerate Per Node | sine_0_6 - rapture
Number of containers violating FPS per Node | sine_0_6 - rapture
Num Containers
node 3
CPU utilization Per Node | sine_0_6 - rapture
% CPU utilization
Memory utilization Per Node | sine_0_6 - rapture
% RAM utilization
GPU utilization Per Node | sine_0_6 - rapture
% GPU utilization
Video Memory utilization Per Node | sine_0_6 - rapture
% VRAM utilization

B

C/R Experiment

Table B.1: List of hooked functions for Glxgears

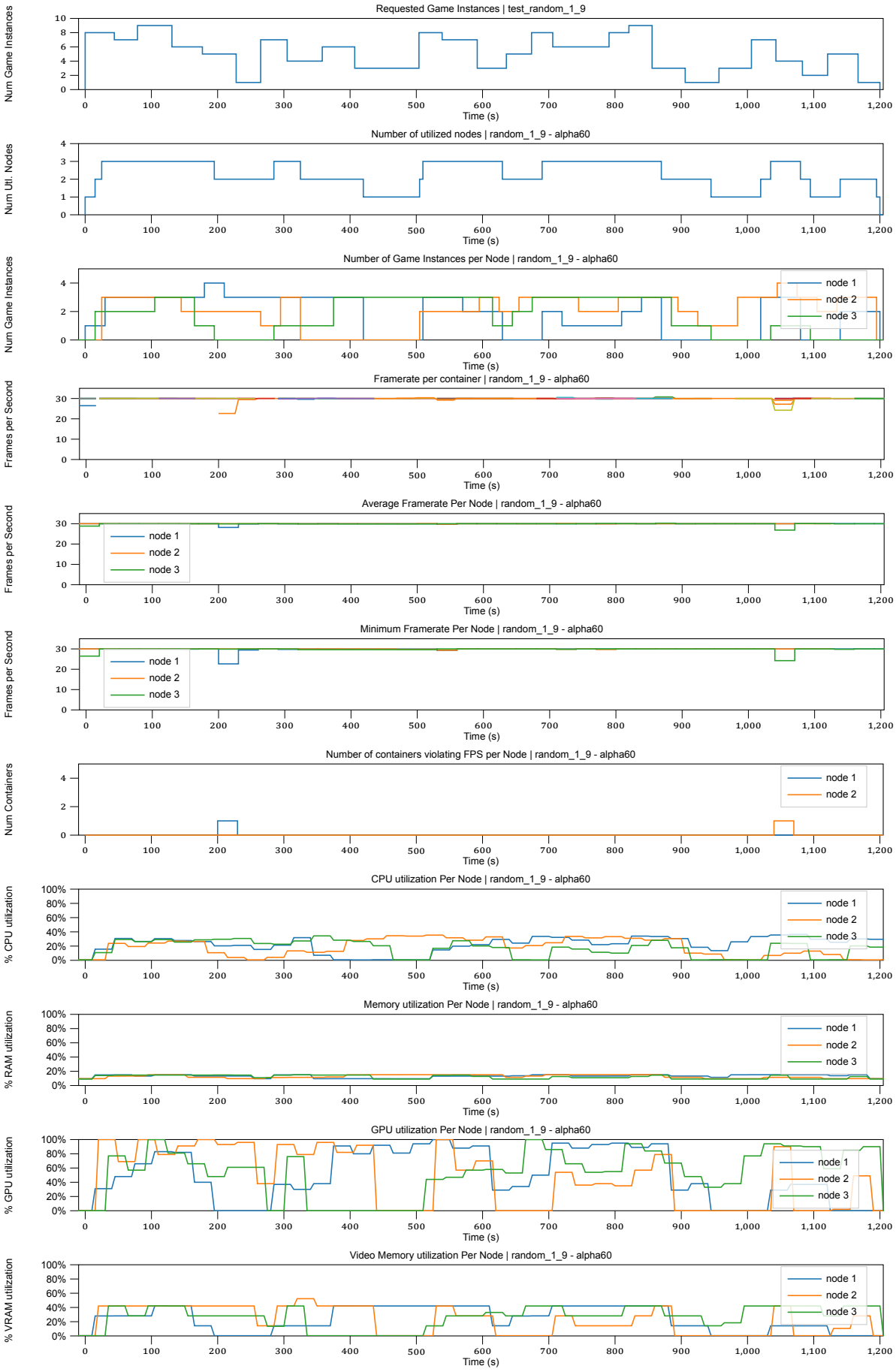
Function	Return type
XDefaultScreen(display)	Int
XDisplayWidth(display, screen_number)	Int
XDisplayHeight(display, screen_number)	Int
XRootWindow(display, screen_number)	Window
XOpenDisplay(display_name)	Display*
XCreateColormap(display, w, visual, alloc)	Colormap
XCreateWindow(display, parent, x, y, width, height, border_width, depth, class, visual, valuemask, attributes)	Window
XSetNormalHints(display, w, &hints)	Int
XSetStandardProperties(display, w, &window_name, &icon_name, icon_pixmap, (char **)NULL, argc, &hints)	Int
XMapWindow(display, w)	Int
XPending(display)	Int
XNextEvent(display, result)	Int
XDestroyWindow(display, w)	Int
glGetString(name)	const GLubyte*
glLightfv(light, pname, params)	-
glEnable(cap)	-
glGenLists(range)	GLuint result
glNewList(list, mode)	-
glMaterialfv(face, pname, params)	-
glEndList()	-
glShadeModel(mode);	-
glNormal3f(nx, ny, nz)	-
glBegin(mode)	-
glVertex3f(x, y, z)	-
glEnd()	-
glViewport(x, y, width, height)	-
glLoadIdentity()	-
glFrustum(left, right, bottom, top, near_val, far_val)	-
glTranslatef(x, y, z)	-
glDrawBuffer(mode)	-
glPushMatrix()	-
glPopMatrix()	-
glTranslated(x, y, z)	-
glClear(mask)	-
glRotatef(angle, x, y, z)	-
glCallList(list)	-
glDeleteLists(list, range)	-
glXDestroyContext(dpy, ctx)	-
*glXChooseVisual(dpy, screen, attribList)	XVisualInfo
glXCreateContext(dpy, &vis, shareList, direct)	GLXContext
glXMakeCurrent(dpy, drawable, ctx)	Bool
glXQueryExtensionsString(dpy, screen)	const char*
glXQueryDrawable(dpy, draw, attribute, &value)	-
glXSwapBuffers(dpy, drawable)	-

C

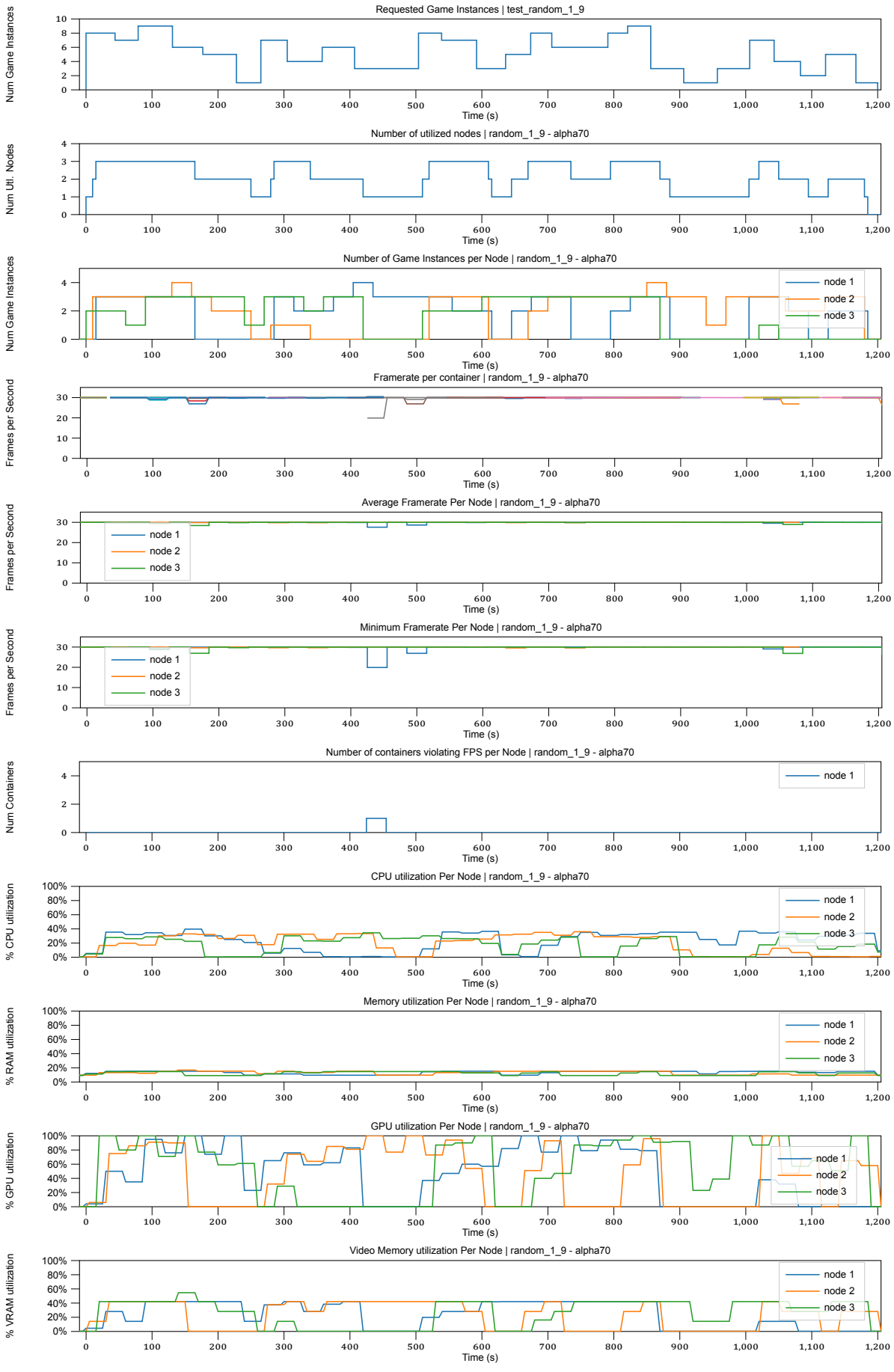
Migration Experiments

C.1. High Load Random Requests

60% utilization threshold

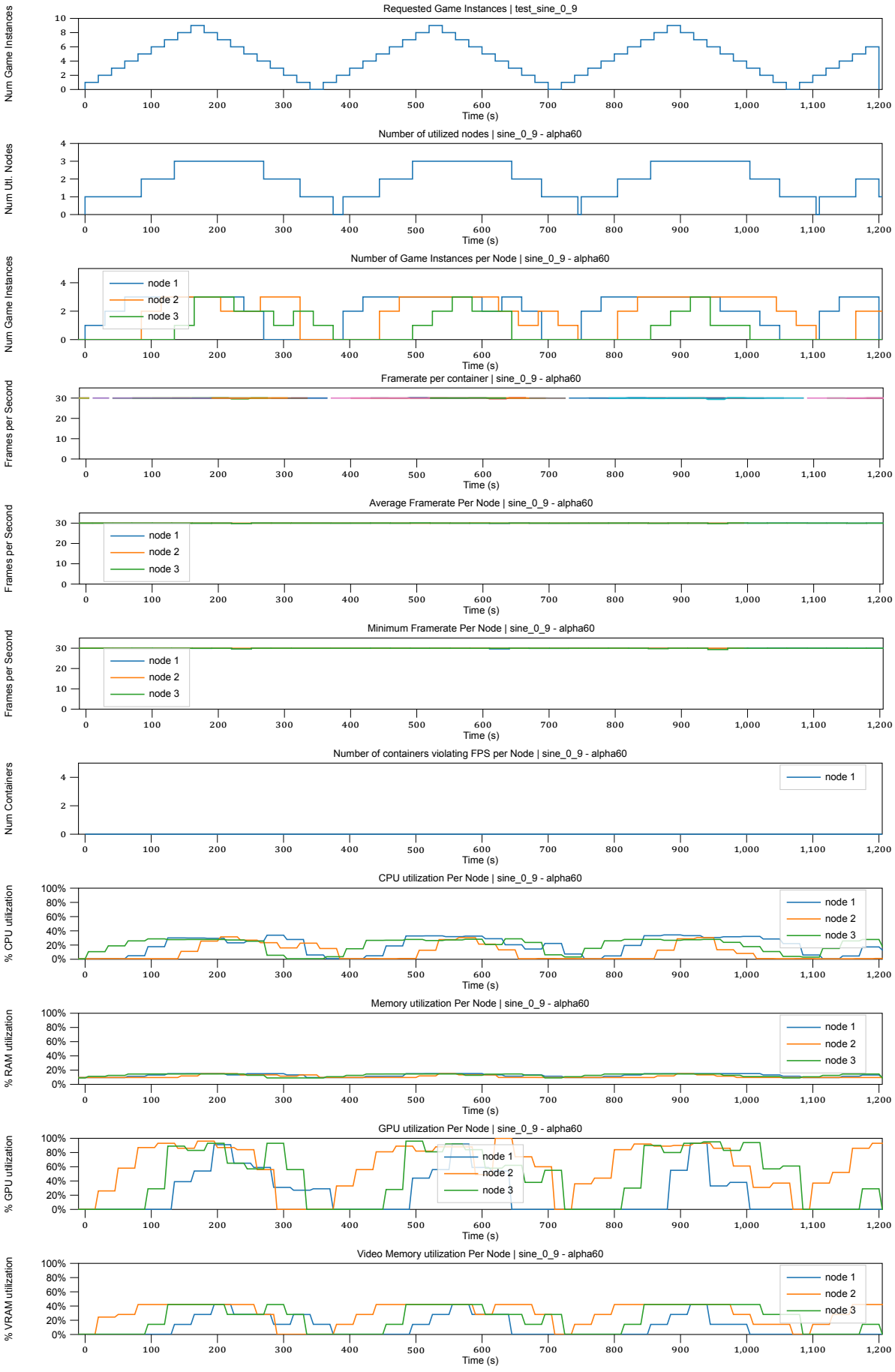


70% utilization threshold



C.2. High Load Sine Requests

60% utilization threshold



70% utilization threshold