Karlsruher Institut für Technologie

Fair scheduling of GPU computation time in virtualized environments

Masterarbeit
von

Tobias Fleig

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl-Inform. Jens Kehne

Bearbeitungszeit: 17. März 2016 – 16. September 2016

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 16. September 2016 . . . .

(Tobias Fleig)

Abstract

Modern GPUs are immensely powerful, highly asynchronous computational accelerators usable for a wide range of applications. Unfortunately, operating systems still treat GPUs like simple devices with predictable response times [28] and are unable to enforce fair sharing of GPU computation time between multiple applications. Additionally, virtualization and cloud-computing became ubiquitous. Most modern GPUs, however, are closed black-box devices, which makes them difficult to virtualize. Previous work on both GPU scheduling and GPU virtualization exists, but does not target the intersection of both topics: GPU scheduling in virtualized environments.

In this thesis, we design, implement and evaluate two fundamentally different approaches for virtualized GPU scheduling. Our primary approach employs a central scheduler in the hypervisor in order to balance the GPU usage of applications over multiple levels of virtualization. We supply the central scheduler with paravirtual hints from the guest to allow the identification of guest tasks. By manipulating the virtual machine's memory mappings of GPU channels, our scheduler is able to separately account and control the GPU usage of individual guest tasks.

Our second approach works fully decentralized and uses separate schedulers on the host and in each virtual machine. Each nested scheduler enforces fair GPU time sharing locally among child tasks.

Based on two existing systems for GPU scheduling and GPU virtualization, we implement a full prototype of the paravirtual approach and a proof of concept of the nested approach.

In the evaluation, we demonstrate our paravirtual prototype's ability to enforce fairness both between virtual machines and between tasks running inside virtual machines at the same time. We measure an average scheduling overhead of only 2.17 %. Additionally, we demonstrate the feasibility of nested GPU scheduling with our decentralized approach. Our proof of concept is able to account and schedule GPU usage in the virtual machine without any help from the hypervisor.

Deutsche Zusammenfassung

Moderne Grafikkarten sind enorm leistungsstarke, weitgehend asynchron arbeitende Rechenbeschleuniger, die für eine Vielzahl von Anwendungen eingesetzt werden. Unglücklicherweise behandeln Betriebssysteme Grafikkarten immer noch wie einfache Zusatzgeräte mit vorhersagbarem Laufzeitverhalten. Infolgedessen sind selbst aktuelle Betriebssysteme nicht in der Lage, die zur Verfügung stehende Rechenzeit auf der Grafikkarte gerecht zwischen allen Anwendungen zu verteilen.

Zudem ist Cloud-Computing mittlerweile allgegenwärtig. Unglücklicherweise ist es sehr schwierig, Grafikkarten gut zu virtualisieren, weil es sich meist um geschlossene Geräte handelt, deren genaue interne Arbeitsweise geheim ist. Zwar existieren sowohl für das Problem der mangelnden Fairness, als auch das der Virtualisierung von Grafikkarten jeweils vielversprechende Ansätze, allerdings beschäftigen sich nur die wenigsten mit der Schnittmenge beider Themen, also Fairness in Cloud-Umgebungen. Darüber hinaus sind uns gar keine Ansätze bekannt, die in der Lage wären die Grafikkartennutzung nicht nur zwischen virtuellen Maschinen, sondern gleichzeitig auch zwischen innerhalb dieser laufenden Anwendungen auszugleichen.

Im Rahmen dieser Arbeit wurden zwei grundlegend unterschiedlich arbeitende Systeme zum Scheduling von Grafikkarten in Cloud-Umgebungen entworfen. Im ersten, zentral arbeitenden System läuft nur ein Scheduler im Hypervisor. Dieser stellt Fairness in allen Virtualisationsschichten gleichzeitig her. Dazu wird der Scheduler mittels Paravirtualisierung mit Informationen über Gastprozesse in virtuellen Maschinen versorgt. Diese Informationen erlauben es, Speichermappings der virtuellen Maschine von Kontrollkanälen der Grafikkarte so zu manipulieren, dass die Grafikkartennutzung jedes Gastprozesses einzeln überwacht und gesteuert werden kann.

Der zweite Ansatz arbeitet mit mehreren verteilten Schedulern, von denen je einer im Hypervisor und in jeder virtuellen Maschine läuft. Die Scheduler sind dabei lediglich dafür zuständig, lokal Fairness zwischen Unterprozessen ihrer Maschine herzustellen.

In der Implementierung wird auf Basis zweier existierender Systeme zur Virtualisierung und zum Scheduling von Grafikkarten der zentral arbeitende Ansatz vollständig implementiert. Dieser ist in der Lage, eine gerechte Verteilung der Grafikkarte gleichzeitig innerhalb des Hypervisors und innerhalb der virtuellen Maschinen zu erreichen. Zudem wurde ein Prototyp des dezentralen Ansatzes implementiert.

In der Evaluierung werden die Fähigkeiten des zentralen Schedulers untersucht und eine vom Scheduling verursachte Verlangsamung von lediglich 2.17 % gemessen. Darüber hinaus demonstriert der dezentral arbeitende Prototyp, dass es auch möglich ist, eine faire Verteilung der Grafikkarte rein mit Software zu erreichen, die innerhalb der virtuellen Maschine läuft.

Contents

1 Introduction

The computational power demands of today's *High Performance Computing* (HPC) applications are enormous. Such applications include scientific simulations [19], weather forecasting [43], and calculations for quantum chemistry [49]. Traditionally, increasing computational demands were answered by raising clock frequencies and adding more general-purpose CPU cores. Since neither approach is viable any longer [5, 12], computing clusters, cloud computing, and dedicated accelerators became widely used.

Today, GPUs are the most popular accelerator for multiple reasons: First, modern GPUs contain thousands [46] of processor cores optimized for parallel floating-point calculations, resulting in a performance increase by multiple orders of magnitude, compared to CPUs [25, 26]. Second, frameworks like NVIDIA's *CUDA* allow flexible programming of these high-performance devices. Third, compared to other accelerators, being a mass-produced consumer article makes GPUs comparatively cheap.

Running computing tasks on short-term rented, virtualized systems in the cloud removes hardware cost, which allows for spontaneous allocation of large amount of resources. The great performance gains possible with GPUs makes their addition to existing cloud offerings desirable. Unfortunately, GPUs are difficult to virtualize [46], partially because of their black-box nature: Most contemporary GPUs are closed devices controlled by proprietary drivers, lacking open or standardized interfaces at the hardware level [42]. Still, promising approaches on GPU virtualization exist [10, 15–17, 42, 46].

Like with any other shared resource, it is the operating system's responsibility to safely multiplex the GPU and provide fair access for all applications. While GPUs evolved and now support concurrent access, operating systems are stuck in the past and still handle GPUs like simple devices with predictable and bounded runtimes [28]. The flexible programming models of modern GPUs, however, allow for computations whose runtimes are neither predictable nor necessarily bounded. Similar to the situation in virtualization, introducing proper, OS-controlled scheduling is difficult because GPUs are complex devices and not much publicly available documentation exists. Again, previous work overcame many of those obstacles and provides valuable insights [4, 18, 25, 26, 41].

While most approaches deal with either virtualization or scheduling exclusively, some [18, 26] target GPU scheduling between virtual machines. None of the previous approaches, however, allow for GPU scheduling between applications running *inside* virtual machines. Yet such scheduling is desperately required: Today, running different GPU applications in parallel in a virtual machine often results in one application getting the majority of GPU computation time.

Our work aims at achieving fair scheduling of GPU computation time, simultaneously both between and inside virtual machines. We use two existing systems for GPU scheduling (NEON [28]) and GPU virtualization (LoGV [16]) as a basis. As an intermediate step, we combine both systems to allow scheduling between entire virtual machines. To also allow scheduling inside virtual machines, we consider two fundamentally different possible approaches: First, a paravirtual approach, which supplies the host scheduler with information about resource allocation by guest tasks. Second, a nested approach in which each guest runs its own scheduler.

We discuss different design aspects of both approaches in detail and build a full prototype of the paravirtual approach. In this approach, the guest system sends paravirtual hints about its child tasks to the host, which allows for separate accounting of their GPU usage. The main challenges in this approach include changing NEON's accounting to be based on individual GPU command submission channels rather than host tasks and passing the required information from the guest to the scheduler in the host. As a secondary goal, we also create a proof of concept implementation of the nested approach.

We measure introduced overhead and scheduling effectiveness in a detailed evaluation. Here, we show that our paravirtual approach is able to enforce fair GPU time sharing both between and inside virtual machines with an average overhead of 2.17 %. A brief evaluation of our nested scheduling implementation also demonstrates this approach's feasibility and recommends further research in this direction.

The remainder of this work is structured as follows: Chapter 2 introduces relevant background information and discusses previous approaches for both GPU virtualization and GPU scheduling. Afterwards, we discuss fundamental design decisions and explore important design aspects in Chapter 3. Specific implementation issues are presented in Chapter 4, followed by a detailed evaluation of our implementations in Chapter 5. We conclude the thesis in Chapter 6 and discuss possible future research.

2 Background & Related Work

This chapter introduces important background information required for the remainder of this work and presents previous research. Section 2.1 introduces *General-purpose computation on graphics processing units* (GPGPU). Important aspects of modern GPU hardware are explained in Section 2.2. Afterwards, related work is presented. Since most previous approaches deal with either GPU virtualization or GPU scheduling exclusively, both topic are discussed individually. Section 2.3 introduces common techniques for GPU virtualization and presents previous attempts. Section 2.4 completes this chapter with a discussion about related work on GPU scheduling.

2.1 GPU as general-purpose processor

Over the past decade, GPUs evolved from pure graphics accelerators to general purpose computation devices used in many HPC applications. Initially, GPUs were only designed for graphics output and their integrated circuits assembled a fixed rendering pipeline [39], with each step designated to one fundamental rendering task like vector transformations or rasterization. As GPUs became more capable, programmable stages were added, which allowed simple programs (*shaders*) to be executed directly on the GPU. Initially, different types of shaders existed for the individual steps of the graphics pipeline. With the addition of more and more shader instructions, running general-purpose algorithms on GPUs became feasible [39].

Contemporary GPUs no longer feature a fixed rendering pipeline or different shader types. Instead, modern GPUs mainly consist of a large number (> 1000 [33]) of simple processor cores, grouped into *streaming multiprocessors*. Compared to modern CPUs, these cores have limited functionality and are not fully independent: In one streaming multiprocessor, all GPU cores must always execute the same instruction, although with different parameters. While this *SIMD*-like [20] mode of operation is not suited for all applications, their enormous level of parallelism allows GPUs to increase performance by orders of magnitude compared to CPUs for well-suited workloads [39]. Today, GPGPU is an integral part of many HPC applications. In context of GPGPU, shaders are called *kernels*.

2.1.1 CUDA & OpenCL

With GPUs becoming suitable to run arbitrary code for GPGPU applications, the need for easy access to the GPUs vast computing power arose. This section introduces the two most popular frameworks for GPGPU scenarios: NVIDIA's *CUDA* and the vendor-independent *Open Computing Language* (OpenCL).

The CUDA [37] framework includes two separate APIs to access the GPU [10], the CUDA *Driver API* and the CUDA *Runtime API*. The CUDA Driver API implements basic primitives to access the GPU that allow a high degree of direct control over the device. To make porting existing applications easier, the Runtime API builds on the Driver API, hides cumbersome details behind an abstraction, and offers *C for CUDA*, a set of extensions to the C programming language that allow direct use of C variables as GPU kernel parameters. CUDA is proprietary and only works on NVIDIA GPUs.

OpenCL [45], by contrast, is an open, vendor-independent standard that not only targets GPUs, but accelerators in general. While OpenCL today is supported by all major GPU vendors, CUDA enjoys greater popularity, which can be attributed to the fact that it was available years earlier. In general, however, performance differences between well-tuned CUDA and OpenCL implementations on the same device are negligible [13].

2.2 GPU hardware

Modern GPUs are complex, asynchronous, and highly parallel accelerators. They feature a large number of streaming multiprocessors, an advanced memory subsystem, a multi-level cache infrastructure [32], and dedicated device memory. This memory often has even lower response times and higher bandwidth than system RAM.

Contemporary GPUs can be used by multiple applications concurrently. In order to allow such sharing safely, GPUs feature a *Memory Management Unit* (MMU) [16]. By creating separate virtual address spaces for each application, the device driver restricts one application's kernels to its own memory. To reduce data transfer overheads, the GPU's memory system allows mapping system RAM into device memory, and vice-versa.

The internals of most contemporary GPUs are a carefully guarded secret [29]. Often, the vendors only provide proprietary drivers that cannot be altered. This blackbox nature of modern GPUs makes it difficult to use them in different ways than those envisioned by the device manufacturers. Nevertheless, reverse-engineering efforts [11] of multiple parties have led to significant discoveries, which allowed the creation of open-source GPU drivers [31, 40].

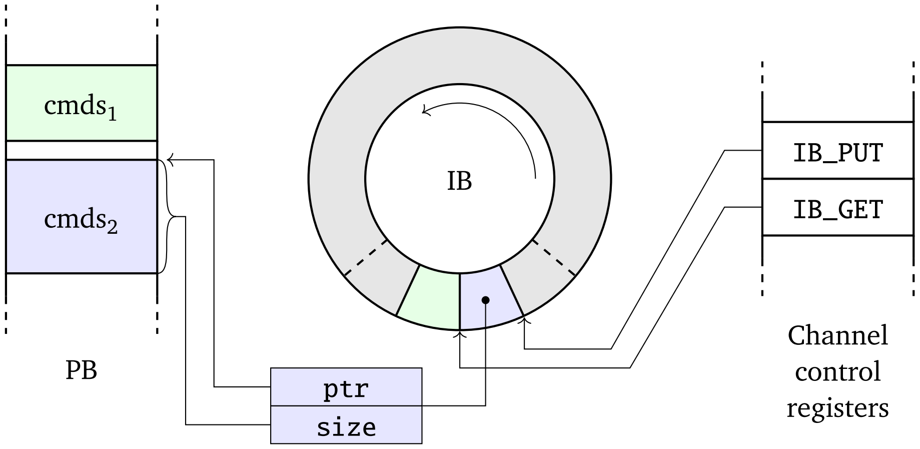


Figure 2.1: GPU command submission channel. The application writes commands required for kernel execution into the *Push Buffer* (PB). Information about the position and size of the entry in the PB are placed in the *Indirect Buffer* (IB). The GPU's channel control registers contain two pointers: `IB_PUT`, which points to the position after the last submitted kernel, and `IB_GET`, which points after the last computed kernel. PB, IB, and the channel control registers are all memory-mapped into the application's address space.

2.2.1 Command submission channels

Traditionally, a driver in the kernel has exclusive control over an attached device. While routing every device access through the kernel works well for slow I/O devices, this approach is unsuited for devices with sub-microsecond command latencies, like GPUs. Calling the device driver requires expensive context switches, which adds too much overhead if performed for every GPU call.

Therefore, today's GPUs offer *command submission channels*, which allow direct device access by userland applications [25]. From the applications point of view, a command submission channel consist of multiple memory-mapped areas. By writing to these, the application can submit work to the GPU without any context switches. Because every command submission channel belongs to one of the GPU's virtual address spaces [16], such direct access can be allowed without endangering memory isolation. It is the responsibility of the kernel driver to correctly set up channels and memory mappings.

Figure 2.1 depicts the structure of a command submission channel. Every channel consists of two memory areas, each mapped into system memory, and memory-mapped control registers: The *Push Buffer* (PB), the *Indirect Buffer* (IB), and the *channel control registers*. To submit a kernel, the application first places the GPU

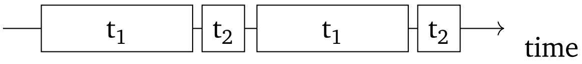


Figure 2.2: GPU internal round-robin scheduling. The GPU executes two kernels from each task. Since the runtimes are different, t_1 gets a significantly larger share of computation time than t_2, which is unfair.

commands required for kernel execution in the Push Buffer. Next, the application creates a corresponding entry in the Indirect Buffer. This buffer is organized as a ring: Entries have a fixed data layout and are addressed by their position in the buffer. The purpose of an entry in the Indirect Buffer is to inform the GPU about the position and size of the commands in the Push Buffer. Kernel execution is controlled by two values in the channel's control register, *IB_PUT*, and *IB_GET*. Both contain index values that point to the Indirect Buffer. IB_PUT points to the index directly after the most recently submitted kernel. After placing an entry in the Indirect Buffer, the application updates this register to trigger computation. IB_GET holds the index of the last computed kernel. The GPU updates this entry to inform the application about completed computations.

2.2.2 Execution model & Concurrency

When multiple applications access the GPU concurrently, or one application uses more than one channel, the GPU has to choose which kernel to execute next. Today's GPUs use a very simple policy for this decision: Round-robin [28]. The GPU iterates over all channels that have uncomputed kernels available and executes one kernel each. Unfortunately, as Figure 2.2 illustrates, this policy is inherently unfair [18, 25, 41]. Applications with longer running kernels get a greater share of GPU computation time. Besides being unfair, this policy is even easily abused: It is beneficial for applications to merge multiple kernels together into one longer running kernel to get more computation time. For malicious applications, it is even possible to create arbitrarily long running kernels, resulting in a *denial of service* [11].

In traditional CPU scheduling, this problem can be trivially solved by introducing a scheduling policy based on *preemption*. Unfortunately, contemporary GPUs do not support preemption of running kernels [28, 41]. Hence, once work on a kernel has started, the computation cannot be interrupted and must run until completion. Although the GPU driver is able to send a special signal to the GPU that causes an execution abort, such an abort does not help with scheduling because execution cannot resume afterwards.

To some degree, modern GPUs are also able to run multiple kernels concurrently [18]. However, this is quite limited: All kernels must belong to the same address space [34]. Furthermore, because kernels from the same channel may depend on the results from previous computations, GPUs will only execute kernels from different channels in parallel [47]. The actual degree of concurrency also depends on the resource utilization of the running kernels. Unfortunately, due to the GPUs black-box nature, the exact mechanics of this mechanism are unknown [11].

The GPU does not generate an interrupt upon kernel completion. This is deliberate, because applications often submit multiple kernels *back-to-back* and are not interested in intermediate results. Not interrupting the CPU after every kernel minimizes overhead. For applications that require knowledge about kernel completion, two possibilities exist: First, the application can poll the IB_GET register and wait for the value to increase. This strategy works, but constant polling also creates CPU overhead in the application. Second, the application can attach a special command to a kernels entry in the Push Buffer, that generates an explicit interrupt [25].

2.3 GPU virtualization

This section presents prior work on GPU virtualization. Since many of the presented approaches use similar strategies, we start with a general overview of GPU virtualization techniques in Section 2.3.1. Afterwards, we briefly discuss the presented approaches, beginning in Section 2.3.2.

2.3.1 Virtualization techniques

GPUs are used for general-purpose computing because of their vast computing power for certain workloads [10, 16, 17, 25, 26, 41, 42]. Consequently, one of the main objectives of GPU virtualization is to keep any introduced overhead low, allowing virtualized guests to achieve near-native performance. At the same time, virtualization systems should be able to multiplex the device, which requires providing basic guarantees like isolation.

A trivial solution is *pass-through*, where one single guest gets full access to the device. While this technique avoids almost all overhead, it also prohibits sharing, since GPUs are not designed to be accessed by more than one host. If sharing the GPU is important, one can use *API remoting*, a technique in which the device driver in the guest is replaced by a stub library. This library simply forwards all calls to a privileged, host-controlled domain that has exclusive access to the GPU.

Depending on the GPU, it may also be possible to use a form of *paravirtualization*. In paravirtualization, the guest is aware of being virtualized and cooperates with the host to make device access more efficient. If knowledge of being virtualized is not desirable, *full virtualization* can be used. In a fully virtualized setting, guests use normal GPU drivers and access the GPU as if it was directly attached to the system. This requires complex device emulation in the VMM, which often introduces unacceptable overheads.

Individual sections below explore the strengths and weaknesses of each approach in detail.

2.3.1.1 Pass-through

In *pass-through*, the hypervisor grants complete control over the GPU to exactly one VM. Most hypervisors support pass-through for arbitrary devices connected to a well-known bus. This is often used for uncommon devices that are unknown to the hypervisor and therefore cannot be multiplexed. GPUs are not uncommon, but difficult to multiplex, so pass-through can be a viable option for GPUs as well. In general, pass-through grants exclusive control, preventing any multiplexing by the hypervisor. A passed-through device can be used by the guest like any device directly connected to the system. Hypervisors often implement pass-through by memory-mapping a particular device directly into the guest. With pass-through, small or even non-existent overheads can be achieved [46].

2.3.1.2 API remoting

Unlike other virtualization techniques, *API remoting* does not grant the guest any kind of access to the device itself. Instead, API remoting allows the guest to use the device by calling functions of a specific API. These calls are forwarded to a privileged domain for execution [46]. Due to its popularity in GPGPU, the CUDA API is a common choice in API remoting.

The library offering the relevant API calls is replaced with a stub library. This stub implements the exact same calls like the replaced API, and can therefore act as a drop-in replacement and does not require any modifications to the applications. When called, the stub library gathers all required data, for example function parameters, and sends them to a privileged domain. The privileged domain has exclusive access to the GPU and executes the requested call on the VMs behalf. It is the responsibility of the receiver of the call in the privileged domain to isolate multiple clients from each other. The returned results are again gathered and sent back to the calling guest, where the stub library unpacks the data and the called API function returns.

One major challenge in implementing API remoting is the design of the communication mechanism used to transfer data. The total delay introduced by data transfers and remote function calls is the primary source of virtualization overhead in API remoting [16]. Therefore, many of the presented approaches below focus almost exclusively on the design of this mechanism.

Implementations of API remoting are often only moderately complex, but suffer from two general drawbacks. First, the overhead introduced by the call forwarding is usually non-negligible. Second, API remoting is limited to the implemented API(s). Applications that use other libraries to access a device cannot be used [46].

2.3.1.3 Paravirtualization

In *paravirtualization*, guest and host cooperate on device virtualization. While this implies the guest has knowledge about using a virtualized device, it usually does not mean guest isolation is a voluntary process and a malicious guest can break the isolation. Cooperation is only required for the guest to properly use the device, while the isolation is still enforced by the host alone.

While it is sometimes also possible to fully emulate devices, such full emulation approaches often suffer from low performance, because the devices and their interfaces were not designed to be virtualized. As an example, a device interface may require additional copying of data to be fully virtualized. In a paravirtualization setting, the VMM can introduce another way of accessing the device that avoids unnecessary copies. Since the introduction of new access methods changes the device's interface, the guest can no longer use normal device drivers. Instead, one must use a modified driver, that includes some knowledge about the way the device is exposed by the hypervisor. Such optimizations often allow device virtualization with very low overhead and near-native performance [16, 46, 48]. However, paravirtualization cannot be used when guest drivers cannot be changed or knowledge in the guest about the virtualization is not desirable.

2.3.1.4 Full virtualization

Guests can access a *fully virtualized* device as if it was directly attached to the system [48]. Since the virtualized device behaves exactly like a real one, normal device drivers can be used and the guest does not have to be aware of or support virtualization. To achieve full virtualization, the hypervisor must precisely emulate the complete device. This works best for simple devices with well known semantics, like ethernet cards. Due to their complexity and proprietary nature, GPUs are very difficult to fully virtualize. While full virtualization systems for GPUs do exist, they often suffer from slow performance due to the high complexity of the emulation and the large number of calls handled by contemporary GPUs [46].

2.3.2 GViM

GViM [17] is a Xen-based virtualization solution for GPUs. GViM employs *API remoting*, and is thus limited to CUDA applications. Special care was taken to optimize the major source of overhead: Copying data from the guest to the privileged domain, where all GPU kernels are executed. A direct mapping solution based on XenStore [8] is used to eliminate most copying in the data path and reduce the overhead to a reasonable amount. Still, GViM must intercept every single CUDA call, serialize it into a packet, and send it to the host for execution.

In contrast to most other approaches in GPU virtualization, GViM actually does deal with scheduling and fairness, although only briefly. A scheduler runs in the privileged domain and decides individually which kernel to execute next. Two policies have been implemented: Round-robin and a credit based system based on Xens credit scheduler [38]. The credit based scheduler uses average GPU kernel runtimes to define GPU ticks, which are then distributed to the VMs based on credits from the Xen scheduler. Since fairness is not the core topic of GViM, the evaluation is rather short. It still shows the credit-based approach to perform better than round-robin.

2.3.3 gVirtuS

gVirtuS [15] aims to be a transparent, VMM independent framework for GPU virtualization. gVirtuS uses API remoting to forward CUDA calls to a privileged domain for execution on the GPU. In gVirtuS, the API remoting runs through an abstraction layer that allows the implementation of different host-guest communication mechanisms, called communicators. By selecting an appropriate communicator, gVirtuS can be adapted to different scenarios and VMMs. The authors initially implemented a TCP/IP based communicator mostly for verification purposes. Since this communicator performed poorly and introduced much overhead, a second one was implemented. This second communicator exposes Unix sockets and is backed by a fast, hypervisor-specific mechanism for efficient host-guest communication. According to the authors, this communicator performed much better and showed a low overall overhead. Unfortunately, the paper does not explain in detail how this communicator works. gVirtuS also does not deal with fairness.

2.3.4 vCUDA

vCUDA [42] is another virtualization solution for GPUs that also relies on API remoting to execute GPGPU workloads in a privileged domain. One important difference, compared to other solutions, is the choice of forwarding mechanism: The authors decided against using a general-purpose host-guest data channel and chose to use a more specialized RPC (remote procedure call) system instead. They argue that such a system is more efficient, as it can exploit knowledge about semantics like data representation for more aggressive optimizations. Initially, vCUDA used XML-RPC [50] exclusively, a general-purpose RPC system that serializes function calls to XML and forwards data via TCP/IP. Since XML-RPC introduced an unacceptable overhead of 43 % to 1600 %, another RPC system was developed: VMRPC [7] uses VMM-managed shared memory to eliminate copy operations, resulting in a reduction of overhead to 1 % to 21 %.

Contrary to most other approaches, vCUDA also supports additional important virtualization features, namely guest suspend, resume, and migration. vCUDA suffers from the same drawbacks as most other approaches: Fairness is never mentioned and vCUDA is limited to applications using the CUDA Runtime API.

2.3.5 rCUDA

rCUDA [10] focusses on remote execution of GPGPU workloads. The authors argue that power can be saved by only adding GPUs to a subset of nodes in a cluster instead of equipping each node with a GPU. To allow all machines to execute kernels, rCUDA implements a stub library, which forwards kernel calls over the network to machines equipped with GPUs. Although based on the CUDA runtime API, rCUDA does not allow usage of the CUDA C extensions, thus requiring application changes for many workloads. As an example, these extensions allow programmers to directly use C variables in CUDA code. Without them, all parameters must be manually transferred to the GPU. The exclusion of the CUDA C extensions is deliberate: Their usage requires the compiler to insert additional calls to undocumented CUDA library functions, which the authors argue cannot be trusted in a distributed environment, because such functions might touch data that must not be shared with other machines (e.g. kernel data). Furthermore, it is argued that lazy call forwarding, which is used by vCUDA, is dangerous, because it may delay error reporting, resulting in altered application behavior.

rCUDA does not deal with scheduling, instead the authors rely on the host GPU driver, which, as explained previously, is problematic.

2.3.6 GPUvm

GPUVM [46] is a virtualization system for NVIDIA GPUs based on Xen. The authors implemented both a paravirtual and a full virtualized version of GPUVM, which allows to compare the two approaches. In full virtualization mode, a complete GPU is emulated for each client. This allows guests to use an unmodified GPU driver. Guest accesses to the GPU are intercepted and emulated in the hypervisor. The usage of *shadow channels* allows each guest to use the full set of the GPU's channels for itself. Similar to many other approaches, the paravirtualization approach translates GPU accesses to hypercalls. To reduce the overhead introduced by hypercalls, the guest is allowed to batch multiple GPU accesses into a single hypercall.

The evaluation reveals unacceptable overheads in full virtualization mode, with multiple benchmarks showing a 80-fold increased runtime compared to native execution. In paravirtual mode, GPUVM performs better, with a maximum application slowdown by a factor of three.

2.3.7 gVirt / Intel GVT-g

GVIRT [48] is a virtualization solution developed by Intel. While primarily targeting Intel GPUs, the authors claim their approach can also be used to virtualize GPUs by other vendors. GVIRT employs *mediated pass-through*, a technique that does not require interception of every GPU access. Instead, GVIRT handles only a subset of important calls. Once a context is established, GVIRT grants the guest partially unrestricted access to achieve near-native performance. Like GPUVM, GVIRT implements full virtualization, allowing unmodified GPU drivers to run in guests. GVIRT also deals with GPU scheduling: VMs get full access for the duration of coarse-grained time slices (16 ms). Since kernel runtimes cannot be predicted and kernel execution is non-preemptive, GVIRT needs to wait for all kernels to complete before switching GPU access to another guest. To prevent guests from exploiting this to overuse their time slices, GVIRT tracks kernel submissions and places a limit on the number of kernels a guest is allowed to queue. Unfortunately, the authors do not explain how this tracking is realized and if costly command interception is required for it.

Intel's GVT-G is the "production-ready", still open-source [22, 23] version of GVIRT offered for Intel GPUs.

2.3.8 NVIDIA GRID & AMD Multiuser GPU

NVIDIA GRID [36] & AMD Multiuser GPU [2] are virtualization solutions offered by the hardware vendors. Unfortunately, not many details are known, other than the information offered in primary sources on the web. Both solutions are closed-source and only work with “professional” server-grade GPUs and certified hardware platforms. NVIDIA claims to allow VMs to access the GPU unobstructed by the hypervisor [35], which hints at some form of *mediated pass-through*, like it was implemented in GVIRT. Similar claims are made by AMD [3]. Furthermore, primary sources for both solutions hint at some kind of fairness mechanism, but it is not explained if this relates to GPU memory or computation time and whether the shares must be statically assigned beforehand.

2.3.9 LoGV

LOGV [16] uses the MMU of modern GPUs to implement a paravirtual virtualization approach. The MMU was originally introduced to allow concurrent GPU access by multiple applications. All memory accesses on the GPU run through the MMU, which confines applications to their respective address spaces. Instead of isolating normal applications, LOGV exploits this mechanism to isolate VMs. Since the isolation is enforced by GPU hardware, LOGV can grant VMs unrestricted access to command submission channels, eliminating most overhead of other virtualization solutions that use API remoting.

The basic architecture of LOGV is as follows: The host uses the *pscnv* (PathScale) driver [40], an open-source driver for NVIDIA GPUs. In the guest, LOGV runs a stub driver that implements the same interface as the host driver. This driver only intercepts and forwards a small subset of calls to the hypervisor, namely those which deal with resource allocation. The hypervisor checks incoming requests to guarantee isolation, and forwards complying request to the host driver for the actual allocation. The hypervisor then maps any returned resources, like channels or GPU memory, into the guest VMs address space and notifies the guest driver. As a result, requesting applications within VMs get direct mappings to GPU resources. These mappings allow guests direct GPU access without any interference by LOGV. Furthermore, since the virtualization happens at the level of command submission channels, LOGV is API-agnostic, because all APIs must use channels at some point. This allows LOGV to run arbitrary, unmodified applications.

The mentioned checks performed by the hypervisor on memory-mapping requests are crucial for correct client isolation. Contemporary GPUs support three kinds of memory accesses between GPU and system memory. In each case, LOGV must only allow mappings that do not violate VM memory isolation. The first kind of memory accesses are mappings of GPU memory into CPU address spaces. These allow the CPU to access GPU memory like any other system RAM. In this case, LOGV

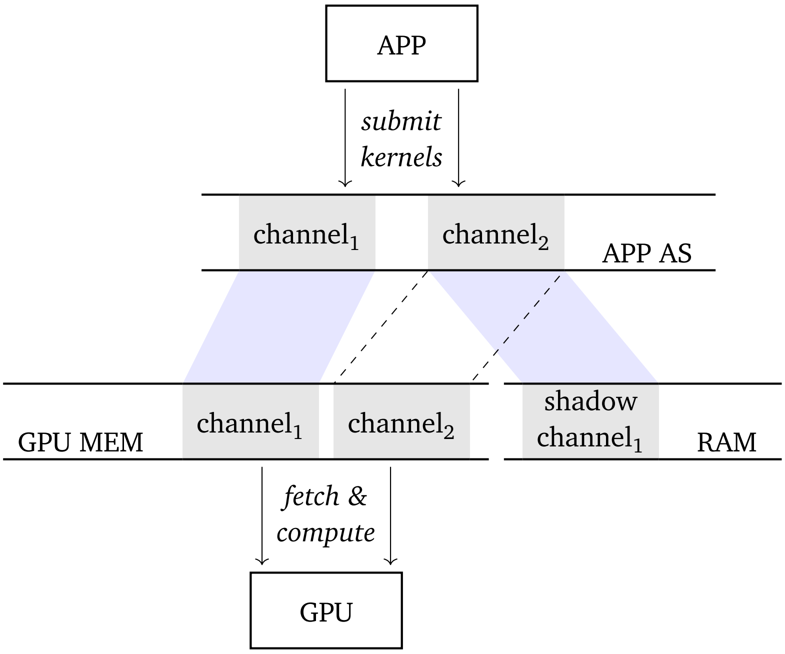


Figure 2.3: Shadow channels: A shadow channel is a transparent modification to an application's channel mapping. Instead of pointing to GPU memory, the shadowed channel points to a kernel buffer in system RAM. When the application submits kernels into a shadowed channel, they do not reach the GPU. Here, channel$_1$ is active with a regular mapping, channel$_2$ is shadowed.

must verify that all of the affected GPU memory originally was allocated by the same VM that is now requesting a mapping. The second kind of memory access is the counterpart to the first one: It allows the GPU to access system memory by mapping it into GPU address spaces. Again, LoGV must verify that all requested system memory belongs to the same VM that created the target GPU address space. Finally, applications can use DMA to transfer data between system and GPU memory. Fortunately, no further verification is required by LoGV in this case, since GPUs implement data transfers as copy operations on memory mappings, which were already verified during their creation.

LoGV supports migration of running virtual machines. Since the GPU operates independent and asynchronous of the CPU, it must be suspended temporarily to allow state migration. To suspend an application's hardware access to the GPU, LoGV temporarily replaces mappings of command submission channels with mappings to *shadow channels*.

Figure 2.3 illustrates the principle of a shadow channel. In this scenario, channel$_1$ resides in GPU memory and is mapped into the application's address space. When the application submits a kernel by writing to the virtual memory address of channel$_1$, the kernel data is written to GPU memory and kernel computation starts. Channel$_2$, however, is currently shadowed: Instead of the GPU channel, an arbitrary kernel buffer is mapped into the application's address space. When the application submits kernels into this channel, they are buffered in system RAM. Shadow channels are fully transparent, the application retains the illusion of direct GPU access, but submitted kernels never reach the GPU. This mechanism allows LoGV to halt the GPU and copy its memory contents to the migration target. As soon as the normal system migration is done, LoGV also copies submitted kernels from the shadow channels to the target, and restores channel access on the target machine.

LoGV was selected as a basis for this work because of its advanced and low-overhead virtualization technique, free availability of source-code and ability to work with NVIDIA GPUs. As detailed in the implementation chapter, this work is based on a rewrite of LoGV, called BLoGV, that uses the proprietary blob NVIDIA driver instead of the PathScale driver for compatibility with more modern GPUs.

2.4 GPU scheduling

This section presents prior approaches for GPU scheduling. The approaches differ greatly in their goals and employed techniques, making it difficult to group similar approaches by categorizing their methods, as it was done in the previous section on GPU virtualization. Most of the presented approaches, however, share one common drawback: The requirement to intercept every GPU access in the scheduler. While this interception can be done in different ways, it will always introduce additional overhead.

2.4.1 GERM

GERM [4] (Graphics Engine Resource Management) is one of the the earliest approaches to GPU scheduling and mainly targets scheduling of graphics workloads. The GERM scheduler resides in a modified GPU driver and has complete control over what is submitted to the GPU. Each process accessing the GPU submits its work to a kernel queue, from which the scheduler chooses the next workload to execute. To track workload runtimes, GERM inserts additional kernels between each two client tasks. The inserted kernels simply increment a host-readable device register, which indicates completion of the previous task. Based on these measurements, a per-process average workload execution time is calculated and maintained. The runtimes of future workloads are then predicted based on the average execution time and the workloads size in bytes and number of vertices. This information is used to run a weighted round-robin scheduling policy.

The evaluation shows that GERM is able to enforce fairness with many graphics-related applications. As soon as GPGPU workloads are added, however, the workload runtime prediction reaches its limits and GERM is no longer able to enforce equal shares of GPU computation time. The main reasons for this are asynchronous data transfers of GPGPU workloads, which GERM does not account for, and that the size of workloads in bytes does not correlate well with execution time for GPGPU applications.

The additional overhead introduced by GERM's scheduler depends on the number of GPU calls an application sends. For GPGPU workloads, the authors measured 1 % to 9 % overhead. For complex graphics applications, which issue a higher number of GPU calls, the overhead was 4 % to 57 %.

2.4.2 PTask

PTask [41] is a set of abstractions that aim at improving the management of GPUs in operating systems. PTask does not target scheduling exclusively. Instead, it is a "big picture" approach to treat GPUs more similar to CPUs in general, also fixing a number of other problems, like GPU availability for OS tasks and OS-based optimization of dataflow. The authors propose a graph, in which nodes represent compute tasks (CPU and GPU) and edges denote memory copy operations. As an example, the operating system can use this graph to automatically eliminate unnecessary copy operations, if two consecutive computation steps both are performed on the GPU.

In PTask, GPU scheduling is ideally no longer controlled by black-box GPU drivers. Instead, the operating system has full control over kernels submitted to the GPU, which allows the OS to enforce fairness and scheduling priorities. In reality, however, the actual kernel submission to the GPU is still controlled by closed-source drivers, so the presented Linux prototype of PTask requires application cooperation in form of explicit additional syscalls before kernel submission. This is an obvious drawback of the presented prototype: GPU scheduling relies on cooperation and is not enforceable by the OS. The authors point out that this is only a limitation of their prototype, based on the way the proprietary drivers work and not a general problem with their approach. However, the paper does not explain how a better implementation enforces fairness without introducing overhead by intercepting GPU accesses in the operating system.

The prototype implements a non-work-conserving bucket-based scheduling algorithm with support for process priorities. A short evaluation shows PTASK to be able to enforce scheduling priorities and fairness. Unfortunately, no measurement of introduced overhead was conducted.

2.4.3 Pegasus

PEGASUS aims at coordinated scheduling for both CPUs and accelerators like GPUs in heterogeneous systems. PEGASUS is one of the rare approaches that deal with both scheduling and virtualization. The virtualization mechanism is not new, it was previously published by the same authors under the name of GVIM [17], which was discussed above in Section 2.3.2. Since the basic virtualization technique was not changed from GVIM, this section focusses on scheduling. In PEGASUS, accelerators like GPUs are first-class schedulable entities, which allows for coordinated scheduling of CPU and GPU computations. The authors argue that such coordination is required to achieve low-latency response times in applications that use both CPU and GPU extensively for their computation. Incoming workloads are queued by an accelerator-specific scheduler that runs in a privileged domain and has complete control over what is submitted to the accelerator. This scheduler also has access to the hypervisors CPU scheduling data, allowing policies to coordinate GPU and CPU processing. As an example, one policy tries to always schedule CPU and accelerators together, thus eliminating queue delays.

To achieve full control over what is submitted to the GPU, the scheduler intercepts every CUDA call. Such interceptions usually result in expensive context-switches or VM exits, introducing overhead. The same holds for API remoting, the employed virtualization technique.

The evaluation hints at a substantial overhead introduced for every CUDA call, with the resulting overhead primarily depending on the number of CUDA calls made.

2.4.4 TimeGraph

TIMEGRAPH [25] attempts to schedule graphics workloads in real-time environments. The authors argue that applications like video players must be prioritized over background work in order to guarantee smooth playback. Like GERM, TIMEGRAPH places the scheduler in a modified GPU driver that controls kernel submissions to the GPU. The scheduler attempts to predict execution times of incoming work in order to improve scheduling. Contrary to the technique used in GERM, this prediction is not only based on one running average kernel execution time per application. Instead, TimeGraph tries to improve the prediction by maintaining averages for multiple sets of GPU command workload sizes independently.

Real-time constraints can be enforced to variable degrees by choosing between two scheduling policies and two reservation policies, which differ in strictness of scheduling priority enforcement, number of produced GPU context switches and prediction usage.

The evaluation compares the different policies and shows that TimeGraph can be used to effectively enforce priorities for GPU-accessing tasks. Enabling scheduling for all applications, however, introduces an overhead of 17 % to 28 %, because every command submission must be intercepted.

2.4.5 Gdev

Gdev [26] aims at improving GPU resource management in the operating system. Like PTask, multiple aspects of resource management are discussed, including memory swapping, shared memory, allowing the OS to use the GPU, and scheduling. The main author previously worked on TimeGraph, which is why the basic scheduling mechanism is similar. However, while TimeGraph intercepts GPU calls at the command level, Gdev works on the API level, intercepting calls to the kernel-based Gdev API. This new API allows usage of new functionality like shared memory on the GPU. To avoid breaking existing applications, a wrapper library was implemented that translates CUDA (Driver API) calls to Gdev API calls. The authors argue this approach is superior to intercepting calls at the command level, because it reduces the number of required interceptions, thus limiting the introduced overhead. However, this approach is no longer API agnostic. Only applications that use the Gdev API, or an API for which a wrapper exists, such as CUDA Driver API, are able to work with Gdev.

Virtualization is mentioned briefly, but is limited to splitting the GPU into multiple logical GPUs. While this provides GPU multiplexing and performance isolation, it is not a full virtualization solution and cannot grant GPU access to virtual machines.

2.4.6 NEON

NEON [28] is a recent approach to GPU scheduling that places a stronger focus on the capabilities of modern GPUs than previous approaches. The authors acknowledge that modern GPUs feature a highly efficient internal task switching, which allows the GPU to be used by multiple processes in parallel. Unfortunately, most previous

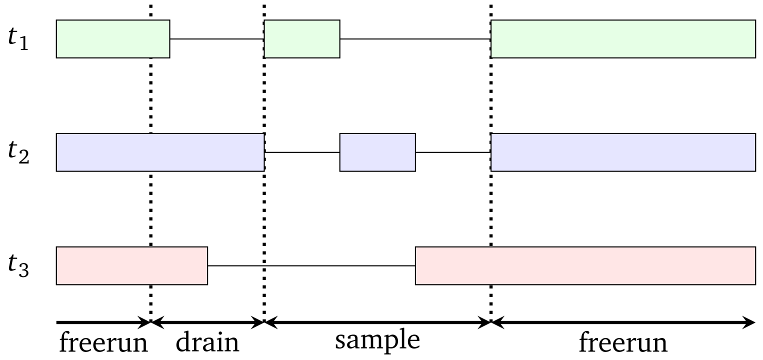


Figure 2.4: NEON's phases. NEON periodically switches between freerun and sampling. Because sampling requires an empty GPU to profile each task, a draining phase is inserted beforehand

scheduling approaches grant exclusive access to single tasks, preventing concurrency and therefore artificially limiting the potential of modern GPUs.

Furthermore, the GPU delivers microsecond-level request latencies [28], which requires direct device access for user tasks to avoid costly crossings of the user-kernel boundary. This direct access is a common problem for scheduling systems, which need to somehow measure the GPU usage of each application to make informed scheduling decisions. Most previous approaches measure GPU usage by intercepting GPU calls in the operating system. While this allows fine-grained accounting, every interception requires a context switch, which re-introduces substantial overhead.

To limit overhead, NEON employs a trade-off, called *disengaged* scheduling. In disengaged scheduling, the scheduler periodically intercepts calls to track GPU usage, but also features a freerun phase, where applications have unobstructed access to the GPU, which allows the GPU's internal context switching to work.

Figure 2.4 illustrates how GPU time accounting works in NEON. After each freerun phase, NEON empties the kernel submission queues by intercepting and blocking all new kernel submissions. NEON then waits until the GPU finishes computation of all previously submitted kernels. This draining is necessary because NEON requires the GPU to be empty for profiling and kernel execution cannot be preempted. After work on all kernels has completed, the sampling phase begins. Here, NEON unblocks the channels of one single application, effectively granting exclusive GPU access. NEON tracks kernel submissions and runtimes only during this sampling phase. After a configurable time slice, NEON revokes GPU access again and profiles the next task. This process is repeated until profiling of all tasks is done. Before switching to freerun, NEON makes a scheduling decision for each task: Tasks that were found to overuse their fair share of GPU time remain blocked, others regain free access to the GPU.

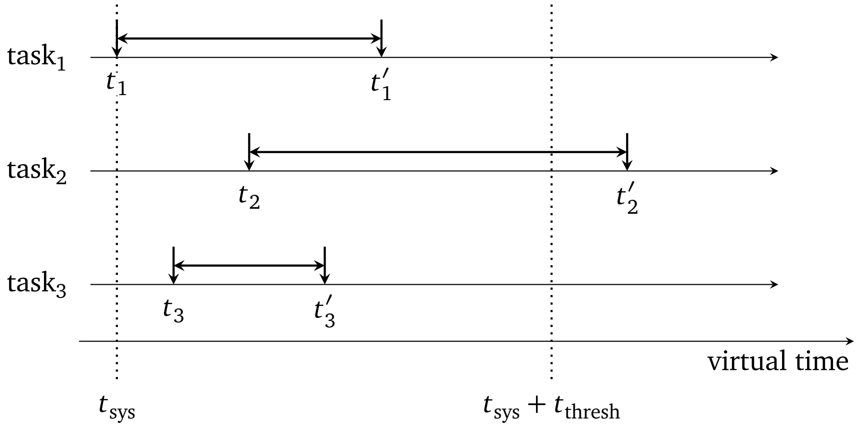


Figure 2.5: Disengaged fair queuing: NEON manages individual GPU time counters for each task and predicts the counter values at the end of an beginning freerun phase by interpolating the GPU utilization measured during sampling. If this prediction exceeds a threshold, the application is not allowed to use the GPU in the upcoming freerun phase.

NEON employs *Disengaged Fair Queuing*, a variant of *Fair Queuing* [9] as scheduling algorithm. Figure 2.5 illustrates how NEON arrives at the scheduling decision. For each task, NEON maintains a counter of GPU time consumed during sampling (t_1, t_2, t_3). Additionally, NEON keeps a global virtual time counter t_{sys} for the whole system. At the beginning of the freerun phase, this system time is set to the minimum GPU time of all active tasks. NEON now increments the individual counter of each task to predict the state at the end of the upcoming freerun phase (t_1', t_2', t_3'). In order to do this, NEON assumes that each task continues to utilize the GPU like it did during sampling. If a task's new counter value is larger than the system time plus a programmable offset t_{thresh}, the task is considered to overuse its fair time share and will remain blocked for the next freerun phase. In this example, task 2 overused its computation time and remains blocked.

Besides Disengaged Fair Queuing, NEON also implements two other scheduling policies: Engaged Timeslice, and Disengaged Timeslice. Both policies use time slices, during which they grant exclusive GPU access to single applications. Engaged Timeslice performs poorly because it intercepts every GPU call. This policy is mostly intended as a baseline for the comparison in the evaluation. Disengaged Timeslice allows unmonitored access for a single application during its time slice. This solves the problem of constant command interception, but is not work-conserving: If the application does not fully utilize the GPU, computation time is wasted.

The evaluation shows that NEON is able to enforce fairness for GPGPU workloads with acceptable overhead. However, the fairness drops significantly when graphics workloads are added. The authors attribute this problem on their lack of knowledge about the behavior of the GPU's internal scheduler when both GPGPU and graphics workloads execute concurrently. They claim that a production-quality version of NEON, written with access to vendor-supplied information about GPU internals, would not have these issues.

NEON does not deal with virtualization. Since QEMU VMs are normal userspace tasks, NEON treats the entire VM like a single application. To achieve scheduling in virtualized environments, we combine the scheduling capabilities of NEON with the virtualization system LoGV.

3 Design

This chapter covers the design process of our work. Section 3.1 starts with a discussion of primary design goals and our definition of fairness for virtualized environments. Afterwards, in Section 3.2, we explore different possible strategies on how to create fairness and select the paravirtual approach for our primary implementation. Section 3.3 discusses all design aspects of this approach in detail. This chapter concludes with a brief discussion of the optional nested approach in Section 3.4.

3.1 Design goals

We choose the following goals for our scheduling system: First, the system must enforce fairness among the children of each scheduling entity, beginning in the hypervisor. Second, entities must not be involved in their own scheduling. Third, the system should introduce as little additional overhead as possible. The remainder of this section discusses explains the reasoning behind these goals.

The overall goal of this work is to provide fair sharing of GPU computation time in virtualized environments. "Fair sharing" in general means that the available computation time is evenly distributed among all applications competing for the GPU. When taking virtualization into account, however, this definition of fairness is no longer appropriate. As an example, a cloud provider who runs multiple virtual machines controlled by clients wants to enforce fair GPU sharing between entire virtual machines first, and within the VMs second. Otherwise, if applications across all virtual machines are treated the same, it is beneficial for clients to run more applications to grab larger shares of GPU time. For this reason, the primary goal of our work is to enforce fair sharing on each virtualization level, beginning in the host.

Figure 3.1 shows how GPU time is distributed with multiple levels of virtualization. In this scenario, the hypervisor runs three tasks, two of which are virtual machines. t_3 hosts another virtual machine. The result is a tree structure. Each internal node's time share is distributed evenly among its children. The numbers below each leaf

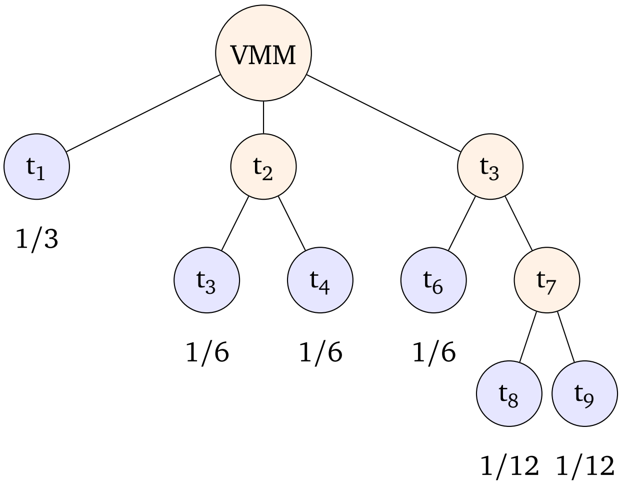


Figure 3.1: Fairness with nested virtualization: Each internal node evenly distributes its available GPU time to its children. Numbers below leafs denote the resulting global GPU time share.

show this leaf's global share of GPU time. This share only depends on the number of children of all ancestors, but not on the number of tasks in other branches.

Enforcing fairness must not depend on cooperation of the virtual machine whose GPU usage is limited. This requirement is a direct result of the cloud scenario. Otherwise, a malicious guest could exploit the cooperation mechanism to obtain larger shares of GPU time. As a general rule, a node must strictly enforce fair time sharing among its children without cooperating with them. In order to achieve such fair sharing, however, the node is allowed to cooperate with its own parent. We can safely allow such cooperation because it is not exploitable for malicious nodes: Since the total GPU usage of a node is always enforced out of the node's reach in its parent, a failure to cooperate only hurts the internal fairness of the node. In the example scenario in Figure 3.1, this means that machine t_3 can cooperate with the hypervisor to balance GPU usage between the children t_6 and t_7. The hypervisor, however, must ensure that virtual machine t_3 as a whole does not overuse its share of GPU time, for which the hypervisor is not allowed to rely on cooperation.

3.2 Design rationale

To the best of our knowledge, scheduling GPU computation time both between and inside virtual machines at the same time has not been attempted before. Previous attempts at GPU scheduling either do not discuss virtual machines, or focus on scheduling between virtual machines only. Due to the lack of experience from prior approaches we considered two fundamentally different designs: Systems with a centralized scheduler running in a privileged domain, and decentralized systems with multiple schedulers. In centralized systems, a single scheduler is responsible for enforcing fairness in all nested virtual machines at the same time. In decentralized systems, on the other hand, each virtual machine features its own scheduler that enforces fairness locally among the virtual machine's children. While it might also be possible to achieve fair scheduling with a solution in which each guest is responsible for its own GPU usage, we did not consider such an approach because of the requirement to always enforce fairness without child cooperation.

Due to the limited time available for this work, we decided to focus first and foremost on the centralized approach and design and implement a fully functional prototype with a centralized scheduler that uses paravirtual hints from the guest in order to enforce fairness within the guest. Our preference for the centralized approach is strongly related to the selection of NEON [28] and LoGV [16] as a basis for our work. We chose these two systems because of their advanced design that works without constant command interception, promising evaluation results, and free availability. With NEON, we already have a central scheduler in the hypervisor that is known to work, so we decided to keep the working basis and augment it to also support scheduling inside virtual machines. Since we were also interested in the decentralized design, we build a limited proof of concept prototype of the decentralized approach as well.

We discuss the design of the centralized approach in detail below beginning in Section 3.3. Afterwards, Section 3.4 introduces the nested approach briefly.

3.3 Paravirtual GPU scheduling

The core observation that leads to the paravirtual approach is the fact that the privileged domain must enforce inter-VM fairness in any case. Since NEON already runs in the hypervisor and enforces inter-VM fairness, it is a natural choice to keep the working system and improve and augment it to satisfy the new requirements.

Figure 3.2 gives an overview of our paravirtual scheduling design. The guest device driver (`nvidia_virt`) forwards GPU calls that deal with channel creation and memory allocation to the VMM. We augment the call that allocates a new command submission channel to include an identifier that allows NEON to distinguish between guest applications. NEON uses this information to build its nested accounting data structures. These data structures are used by NEON's scheduler, which controls

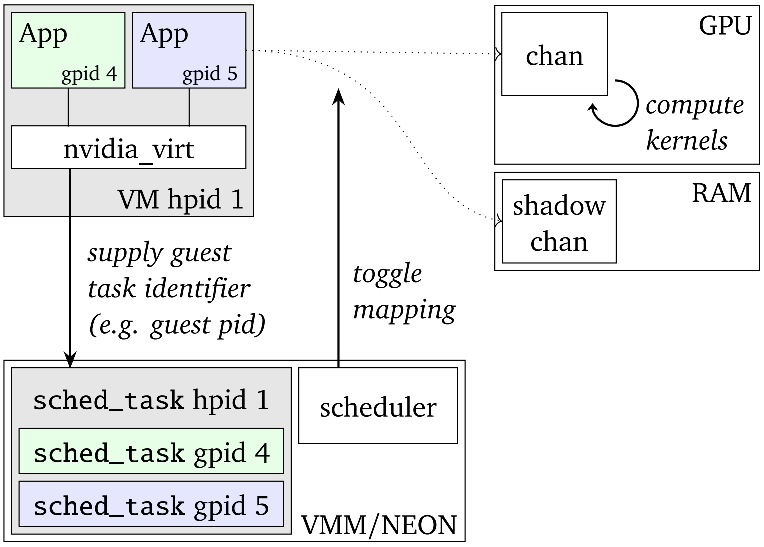


Figure 3.2: Structure of the paravirtual approach: The virtual GPU driver (`nvidia_virt`) sends hints during channel creation, that allow the VMM to create separate scheduling data structures (`sched_task`) for each task in the virtual machine. The `scheduler` switches each guest task's channel mapping with a mapping to a *shadow channel* to control GPU access

GPU access of guest applications by switching their channel mappings between real GPU channels and *shadow channels* that only buffer submitted commands. The following sections describe each component in detail.

3.3.1 Guest driver

In order to enforce fairness among processes running in a virtual machine, NEON must first learn about their existence. It is the responsibility of the VM's virtual GPU driver to provide the necessary information. LoGV's virtual GPU driver forwards all calls that are required for channel allocation and memory management to the hypervisor, and memory-maps the returned resources into the requesting application's address space.

LoGV's virtual machines are based on QEMU, which makes them normal processes alongside all other host processes. Consequently, NEON sees the entire virtual machine as one single task that uses multiple channels. In order to allow for more fine-granular scheduling of tasks running inside the guest, NEON needs to know which channel belongs to which guest application. Figure 3.3 illustrates why NEON cannot easily deduce this information purely from what the hypervisor observes. In

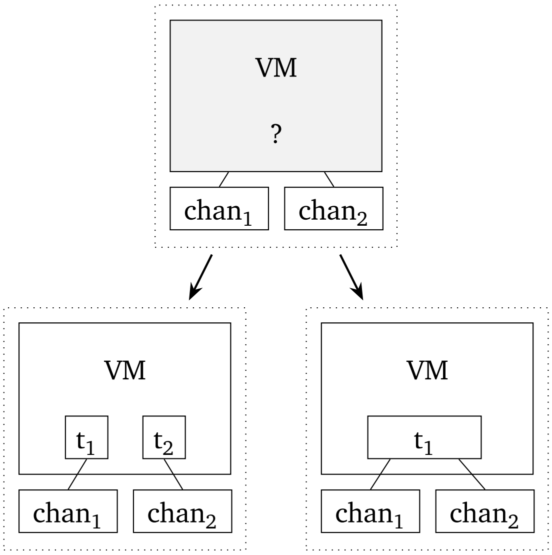


Figure 3.3: Two possible solutions for the observation: “One VM, two channels”. There are either two guest tasks, using one channel each, or just one, which uses both channels. Without additional information, NEON cannot distinguish the two.

the depicted scenario, NEON sees one virtual machine that uses two GPU channels. This observation, however, allows two different conclusions: In one case, only a single task runs inside the virtual machine and uses both channels. In the other case, there really are two tasks running in the VM. For correct scheduling, NEON must be able to distinguish these two cases.

Unfortunately, it is difficult for the hypervisor to obtain the information required to identify guest tasks. Since the guest operating system may implement tasks in a number of different ways, the hypervisor has no easy way of identifying guest processes. We therefore decided to rely on cooperation in form of paravirtual hints. More precisely, the guest’s virtual GPU driver supplies an identifier, such as a process id, when allocating a channel. This allows NEON to distinguish guest tasks.

Solving the problem of guest task identification with paravirtual hints has a number of advantages over other techniques based on VM introspection [30]: First, attaching a guest task identifier is easy to implement for both the guest GPU driver and the hypervisor. Second, this strategy is independent from the way the guest operating system implements tasks. Last, the guest driver can even choose to group GPU channels by something else than guest tasks. The hypervisor will create one scheduling entity for each unique identifier sent by the guest, so this interface allows the guest to arbitrarily group channels to scheduling entities.

Relying on cooperation for guest task identification is safe, because this mechanism cannot be exploited by malicious guests in order to increase their share of GPU time. The fairness between the virtual machine and its siblings is always enforced by the hypervisor without their cooperation. If a VM decides to cease cooperation, only the fairness between its own child tasks will suffer, the rest of the system remains unaffected.

3.3.2 Channel-based GPU access control

In order to provide fairness, our system must be able to prevent applications from submitting work to the GPU. If we detect an application that is overusing its fair share of GPU time, we block that application's access to the GPU temporarily.

One technique that prevents applications from accessing the GPU is to completely stop their execution. The original version of NEON catches channel accesses and blocks the accessing thread. Unfortunately, we cannot use such a simple technique, because our system must be able to target applications that run inside a virtual machine. Since LoGV's virtual machines are based on QEMU, which is a full system simulator, channel accesses come from host processes that are entire *vCPUs*. Blocking the execution of such processes thus does not target a single guest application, but stops a large part of the virtual machine, which is not the desired effect. Instead, we require a mechanism that allows GPU access control on a per-channel basis.

Since influencing the execution of guest tasks from the hypervisor is very difficult, we decide to use LoGV's shadow channels [16] to solve this problem. With shadow channels, we no longer stop the execution of an application completely to prevent GPU access. Instead, we modify the application's mapping of its command submission channel to point to a buffer in system RAM instead of the actual channel provided by the GPU driver. As a result, the application unknowingly writes to the buffer when submitting new work. From the GPU's point of view, the application is idle and no new work arrives. In order to unblock the application, we reset the mapping to point to the real GPU channel again and copy remaining work from the shadow channel to the GPU for immediate computation.

3.3.3 Grouping data structures

NEON's internal data structures were designed with the assumption that all channels used by one application belong to the same scheduling entity. With virtualization, this assumption is no longer valid. A QEMU virtual machine may contain any number of guest processes, and guests may even create their own VMs with additional tasks.

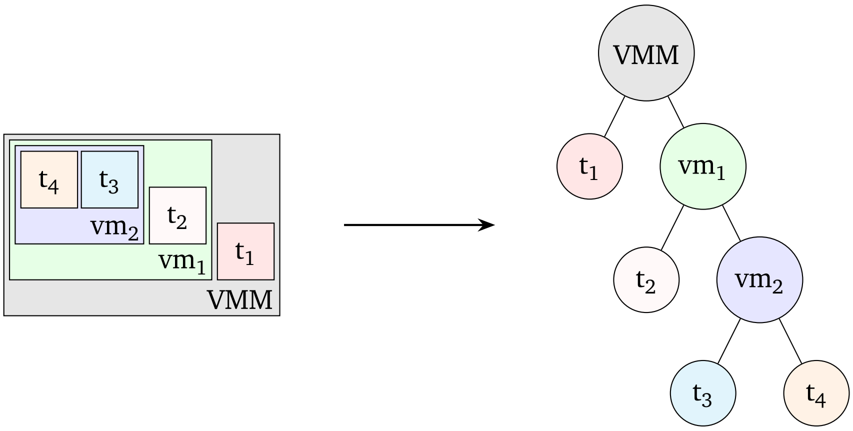


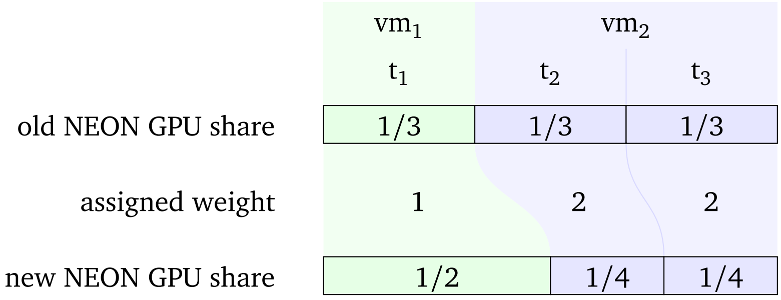
Figure 3.4: Representation of nested virtualization. On each level, the scheduler must divide the available GPU computation time fairly

To properly represent the potentially complex, nested VM structure, we require a tree. Figure 3.4 shows an example: The root node represents the hypervisor. Tasks running directly in the hypervisor domain are children of this root node. When a child task of a virtual machine allocates a channel for the first time, we create a new scheduling entity as a child of the virtual machine's node. On each level, the scheduler must balance the available GPU time between all direct child nodes.

3.3.4 Group-aware scheduling

In order to provide fairness for multiple levels of virtualized systems, we need to adapt NEON's scheduling policy. We run NEON with the *Disengaged Fair Queueing* (DFQ) policy. Like all policies implemented in NEON, DFQ aims at creating fairness among all competing applications. With virtualization, however, it is no longer sufficient to treat all scheduling entities the same. Figure 3.5 shows an example: In this scenario, one application runs inside its own VM, and two applications run inside another virtual machine. NEON's unmodified scheduling algorithm grants one third of the available GPU time to each application. While this partitioning fairly distributes GPU time within vm_2, it violates the hypervisors fairness requirements, because vm_1 gets less computation time than vm_2.

In order to support grouping, we assign weights to each scheduling entity. The weight of a scheduling entity is the inverse of the share of GPU computation time the entity receives, compared to a task running directly on the host. As an example, a scheduling entity with weight w will receive $1/w$ of the GPU time of a non-virtualized task. With nested virtualization, the weight of a scheduling entity can



	vm_1	vm_2	
	t_1	t_2	t_3
old NEON GPU share	1/3	1/3	1/3
assigned weight	1	2	2
new NEON GPU share	1/2	1/4	1/4

Figure 3.5: Two levels of scheduling with Disengaged Fair Queueing. To achieve fairness both between and and inside VMs, tasks are assigned with weights that equal the size of their scheduling group.

be calculated as follows: Given a scheduling entity n, its parent entity p_{n}, and the number of children of an entity e as $c(e)$, the weight is recursively defined as:

$$w(n) = \begin{cases} w(p_n) \cdot c(p_{\text{n}}) & \text{, if } p_{\text{n}} \text{ exists} \\ 1 & \text{, otherwise} \end{cases} \tag{3.1}$$

In other words, all scheduling entities running directly on the host are assigned weight 1. The weight of a scheduling entity not running on the host is the product of the number of children of each of its ancestors. Assigning weights to scheduling entities as defined in Formula 3.1 therefore effectively linearizes the tree structure of scheduling domains, which allows the DFQ scheduler to satisfy all scheduling constraints.

In the situation depicted in Figure 3.5, we assign weight 1 to t_1, and weight 2 to both t_2 and t_3. The scheduler then only allows t_2 and t_3 to run half as long as t_1, which results in the desired split depicted at the bottom of Figure 3.5.

3.4 Nested GPU scheduling

The main argument for the nested approach is that fairness inside virtual machines is not strictly required from the hypervisor's point of view, and therefore should not be its responsibility. In this approach, every virtual machine runs its own scheduler, which only provides fairness locally among child tasks of the virtual machine.

Since the primary focus of our work lies on the paravirtual approach, we only created a nested proof of concept implementation, that does not feature a full design. In the remainder of this section, we briefly introduce how our prototype

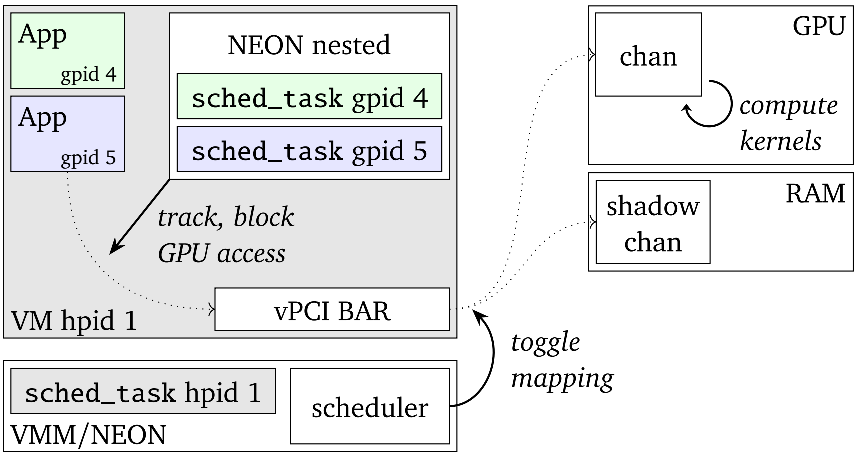


Figure 3.6: Structure of the nested approach: Thy hypervisor accounts and controls the virtual machine as a whole by using shadow channels, similar to the paravirtual approach. Fairness inside the virtual machine is enforced by the nested scheduler, which tracks and controls applications based on their mapping of the virtual PCI BAR device.

works. Afterwards, Section 3.4.1 discusses one of the most important differences between centralized and decentralized approaches: How to organize sampling.

Figure 3.6 shows how our proof of concept works. One scheduler runs in the hypervisor and distributes the GPU time equally between all tasks and virtual machines that run directly in the hypervisor. This scheduler treats virtual machines like normal processes and measures their GPU usage as a whole, without considering internal tasks. Inside each virtual machine runs a modified version of NEON, which is responsible for enforcing fairness for the children of its virtual machine.

LoGV's virtual GPU driver maps command submission channels from the virtual PCI BAR device into application memory. There is no difference between such a mapping created by LoGV and a real mapping created by the NVIDIA driver. Therefore, NEON can manipulate this mapping in the same way as it does in the host to track kernel submissions. Consequently, no modifications to NEON's accounting mechanism are required.

In order to keep the effort manageable, we share as much code as possible with the paravirtual approach. Therefore, the hypervisor also uses shadow channels to temporarily prevent virtual machines from submitting work to the GPU. While this works well, it is not strictly required from a technical point of view. Using a technique that fully blocks the execution of the virtual machine, similar to what the original version of NEON does, is also a possibility.

3.4.1 Nested sampling

Proper scheduling requires precise measurements of resource usage. NEON is based on the assumption that an application's usage of the GPU does not vary greatly during short time intervals. This assumption allows NEON to measure GPU usage only during a short interval in the sampling phase, interpolate the measurements, and then decide to block or unblock an application for the freerun phase.

We identified two different possibilities on how to organize sampling in the nested approach: First, nested schedulers run their own sampling. Second, the hypervisor samples all tasks and provides the results to the guests. For the sake of simplicity, our prototype uses the first, truly decentralized option. However, leaving the sampling to clients also introduces complex scheduling interactions, that need to be addressed.

The core problem with nested sampling is the base assumption of NEON that it is the only scheduler and has complete control over the GPU. When NEON runs in a virtual machine, this is no longer true. During the sampling phase, NEON assumes only the sampled task can access the GPU and all observed utilization is created by one single application. In a nested environment, however, the hypervisor might be in the freerun phase at the same time. In this case, work submitted by applications from other virtual machines also reaches the GPU and disturbs the measurement of the nested scheduler. A similar situation can arise when the nested scheduler tries to sample a task, while the hypervisor blocks GPU access for the whole virtual machine. This situation also results in wrong measurements, because the internal scheduler assumes the kernels of the sampled application run for a very long time, when in reality they just never reach the GPU because the hypervisor redirects them to a shadow channel.

Finding a good strategy to avoid such problems in nested sampling is difficult. We briefly considered some kind of coordination that synchronizes the phases of the schedulers and avoids many problematic interactions. Another core problem with nested sampling, however, cannot be solved by coordination: Sampling GPU usage in each virtual machine separately scales badly. Without exclusive GPU access, NEON's sampling cannot produce reliable results. Consequently, each scheduler needs exclusive GPU time in order to conduct precise measurements. As the number of schedulers grows in more complex scenarios with multiple layers of virtualization, we allocate an increasingly larger share of the total available GPU time for exclusive sampling phases. Such extensive phases of exclusive access, however, are a major source of overhead, which we sought to avoid in the first place.

The second option on how to organize sampling with nested virtualization is to keep all sampling in the hypervisor and export the results to the schedulers running in the virtual machines. We suspect the overhead of this approach to be much lower than for nested sampling, because every task is only sampled once by the hypervisor. Furthermore, no complex interactions can arise when all sampling is performed in one place. Central sampling, however, again requires knowledge of child tasks in the hypervisor, which leads back to a more centralized design like the paravirtual approach.

For our proof of concept implementation, we use uncoordinated nested sampling and manually configure the duration of the sampling phase in the nested scheduler to a fraction of the duration in the hypervisor's scheduler. This strategy is acceptable for our prototype, because we do not need to support larger virtualization scenarios.

4 Implementation

This chapter covers the implementation part of our work. Our primary goal is the creation of a complete prototype of the paravirtual approach that fulfills all design targets discussed in Chapter 3. Furthermore, we aim to demonstrate the feasibility of the nested scheduling approach by creating a proof of concept implementation.

In general, virtualization may be nested. In order to support the cloud scenario, however, building a prototype that enforces fairness with only one layer of virtualization is sufficient. For simplicity, we therefore decide upfront to limit our implementations to one level of virtualization.

The remainder of this chapter is structured as follows: Section 4.1 discusses the selection of the target platform. We describe the implementation of the basic virtualization support for NEON in Section 4.2. The core implementation effort is described in Section 4.3, where we discuss the implementation of the paravirtual approach. Finally, Section 4.4 gives a brief overview over the implementation of the proof of concept for the nested approach.

4.1 Target platform

We use NEON [28] and LoGV [16] as a basis for our implementation. NEON hooks into the NVIDIA GPU driver and parses driver-internal data structures. The original authors of NEON built the necessary parsing logic largely based on information obtained by reverse engineering [11, 29]. Since driver-internal data structures are subject to change without notice, NEON's parsing logic is delicate and breaks easily when other driver versions are used. Consequently, we try to keep our target platform as closely as possible to the one used in the original NEON paper.

Because NEON is tied to the proprietary NVIDIA driver, but LoGV requires the open-source pscnv [40] driver, we cannot use LoGV directly. Instead, we use BLoGV, a rewrite of LoGV that works with the NVIDIA BLOB driver.

The original version of NEON does not feature full support for *Symmetric Multiprocessing* (SMP) [27]. In order to properly run multiple virtual machines, however, we absolutely require SMP. We therefore introduce SMP support in NEON and add the necessary locking.

4.2 Virtualization support

The original version of NEON manipulates page tables and uses the processor's single-stepping debug feature in order to track kernel submissions during the sampling phase. Since our virtual machines use *Second Level Address Translation* (SLAT) [1], NEON's normal page table manipulations do not affect applications running in virtual machines. Similarly, single stepping a particular application in the virtual machine from the hypervisor is more complex.

We discuss our modifications to NEON's page table manipulation mechanism in Section 4.2.1. Afterwards, we describe our technique for single-stepping in Section 4.2.2.

4.2.1 Second Level Address Translation

NEON manipulates page table entries of command submission channels in order to account and control GPU access of applications. By clearing the *present* bit of a channel's page table entry, NEON forces a page fault on the next access by the application, which allows NEON to precisely count channel accesses. On modern platforms, however, manipulating the normal page table is not sufficient to control the GPU usage of virtual machines, because such platforms feature hardware assisted virtualization in form of *Second Level Address Translation* (SLAT). The first implementation step is therefore to add SLAT support to NEON. This is required for both the paravirtual and nested approaches. Intel's implementation of SLAT is called *Extended Page Tables* (EPT).

Figure 4.1 shows the different mappings that play a role in the address resolution of a command submission channel that is in use by an application in a virtual machine. In total, three mappings exist: First, the traditional mapping into QEMU's address space. This mapping is stored in the host operating system. Second, a mapping of the channel into the physical address space of the virtual machine. This mapping is controlled by the hypervisor. Last, a mapping from the guest physical address space into the address space of the guest application. This mapping is controlled by the guest operating system.

In order to resolve a guest-virtual address, the MMU first uses the guest application page table to retrieve a guest-physical address. Afterwards, the MMU consults the EPT to translate the guest-physical to a host-physical address. The original version of NEON, however, only manipulates the normal page table entry on the host, which is not involved in this address resolution. As a result, changes by the original version of NEON do not affect GPU accesses by the virtual machine.

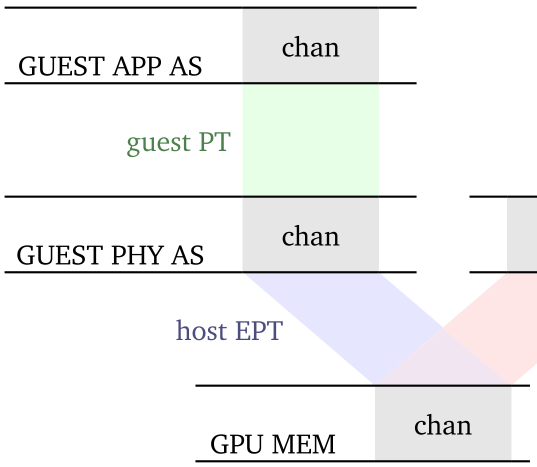


Figure 4.1: Channel mapping with Second Level Address Translation: The channel mapping of normal host processes and QEMU resides in the host page table. KVM copies this entry once into the EPT, which creates a mapping into the guest physical address space. The virtual GPU driver then maps the channel into the guest application's address space. This mapping resides in the guest page tables. The original version of NEON works with the host page table, our new version uses the EPT for VM tasks. A nested scheduler uses the entry in the guest page table.

We port NEON's existing page table manipulation mechanism to work with EPT tables. We add a hook in the Linux kernel in order to detect when KVM creates an EPT entry for a known command submission channel. Since a channel is never used by both host and guest applications at the same time, we only need to manipulate either the host page table or the EPT for NEON's tracking. Therefore, we add a flag to NEON's channel control data structure that indicates if a channel is currently mapped in an EPT entry.

4.2.2 Single stepping

NEON also requires single-stepping in order to track channel usage. After a modified page table entry triggered a page fault, NEON resets the present bit in order to allow the application to continue. At some later point in time, however, NEON needs to clear the present bit again to catch the next channel access. NEON uses the processor's *Trap Flag* (TF) control bit in order to avoid leaving the channel accessible for too long, which can result in missed channel accesses. This flag is originally intended for single-step debugging and causes a context switch back to the operating system after each single application instruction. NEON exploits this functionality by enabling the processor's trap flag along with a channel's present bit. As a result, the application executes exactly one instruction, which is the repetition of the previously failed channel access. Directly afterwards, the trap flag leads to a kernel trap, during which NEON disables the flag and clears the present bit again. This strategy prevents missing any channel accesses.

In the virtualized scenario, however, using the trap flag is non-trivial for two reasons: First, we need to take care to actually single-step the target application that runs inside the virtual machine, and not only execute a single instruction of QEMU or the guest operating system. Second, manipulating the trap flag from the hypervisor can lead to complex interactions when the guest operating system is using this flag at the same time. As an example, this is the case if another GPU scheduler runs inside the virtual machine, as we plan to do in the nested scheduling approach.

Fortunately, Intel's hardware virtualization extensions feature a viable alternative to the trap flag, called the *Monitor Trap Flag* (MTF). This flag allows single-stepping of the virtual machine without any interactions with the VM's trap flag. We therefore added functions to KVM that allow NEON to set the MTF flag at will, and use these functions in order to track accesses to EPT-based channels.

4.3 Implementing paravirtual scheduling

This section covers the implementation of the paravirtual approach. We discuss the main design topics from Section 3.3 in the same order, and explain how we implement each aspect in our prototype. Our implementation of the paravirtual approach is feature-complete and runs stable.

4.3.1 Guest driver

We require paravirtual hints from the guest GPU driver in order to correctly assign allocated channels to guest tasks. Since our prototype only supports one level of virtualization, one single identifier is sufficient to distinguish guest tasks.

A suitable identifier has the following properties: First, the identifier must be unique within its virtual machine. With duplicates, we cannot distinguish tasks. Second, the identifier must not change during the lifetime of the VM. Supporting non-constant identifiers adds unnecessary complexity. One readily available identifier that fulfills both requirements is the guest operating system's process id.

In order to allocate a channel, the CUDA runtime requests multiple memory mappings by sending *HOST_MAP* commands via the *ioctl* [44] channel to the GPU driver. In the virtualized scenario, the virtual GPU driver forwards these calls to the hypervisor. Careful analysis of the parameters of the HOST_MAP command shows that one of the parameters is always zero. Furthermore, changing this parameter to an arbitrary value does not create any measurable effect. Consequently, we decide to use this presumably unused parameter to piggyback our task identifier. We modify BLOGV to insert the guest task id here, before sending off the ioctl as usual. Since NEON already intercepts ioctl calls to track memory mappings, extracting the guest task id from the call is just a simple modification.

4.3.2 Shadow channels

We use shadow channels to prevent tasks inside the virtual machine from sending more work to the GPU. A shadow channel is essentially a kernel memory buffer that is mapped into the application and replaces the mapping of the real command submission channel. When the application writes to the channel, it unknowingly writes to the kernel buffer and the commands never reach the GPU. A command submission channel consists of three major memory areas: The Push Buffer, the Indirect Buffer, and the channel control registers. The control registers contain IB_PUT, an index value that points to the newest entry in the Indirect Buffer. By incrementing IB_PUT, the application notifies the GPU about new entries in the other buffers. See Section 2.2.1 for details.

For NEON's use case, it is sufficient to shadow the channel control register mapping, since the GPU does not perform any computation before the application increments IB_PUT. The application retains its mappings of the Push Buffer and the Indirect Buffer. This is unproblematic, because the GPU ignores any data written there until IB_PUT changes.

Shadow channels should be transparent for the application. In order to maintain this transparency, we need to guarantee a consistent view on the command submission channel, even while enabling or disabling the shadow channel. Maintaining a consistent view requires copying data from the GPU channel into the shadow channel's kernel buffer, so the application reads the same data before and after the switch. The same is true when switching back to the GPU channel later on.

Copying data and modifying the channel mapping are two separate operations, which cannot be performed together atomically. This leads to a race condition: If we change the mapping before copying the data, the application may see outdated values still present at the new mapping target. On the other hand, if we copy the data first and modify the mapping afterwards, the application may update the values within the old mapping again after the copy operation, but before the new mapping is in effect. This data race is present during both switches to and from the shadow channel. However, enabling the shadow channel is trivial, because NEON decides to switch while the application waits in the page fault handler. Hence, the application is already stopped, so the race is unproblematic.

Unshadowing, however, is more complex because it happens asynchronously to application execution, so the application may be actively using the channel during unshadow. To solve this problem, we first set the access permissions of the shadow channel's page to read-only. Subsequent write accesses now result in a page fault, which we catch in the page fault handler. After setting the page read-only, the application can no longer change any data. Thus, we are free to copy IB_PUT back to the GPU. Afterwards, we restore the original channel mapping and allow the application to proceed through the page fault handler.

The switch back to the real GPU channel can take place for two different reasons: First, all applications that did not exceed their fair share of GPU time during sampling regain access when the freerun phase begins. Second, NEON unblocks applications separately in the sampling phase in order to measure their GPU usage. When switching to freerun, unblocking works as described: We switch back the mapping to the GPU channel and copy IB_PUT to the card. Switching to sampling, however, is more complicated, because of the the changed semantics of a shadowed channel compared to blocking an application in the page fault handler. While both techniques prevent further kernels from reaching the GPU, shadow channels still allow the application to submit work. When NEON disables the shadow channel and copies IB_PUT to the GPU, the card immediately starts computing any work that was submitted while the channel was shadowed. This poses a problem for NEON's accounting, which needs to precisely count the number of computed kernels during sampling. The shadow channel, however, contains an unknown number of kernels that will now contribute to the measured runtime, without being accounted for. Figure 4.2 depicts such a situation. Here, the application submitted two kernels to the shadowed channel during the draining phase. When NEON switches to sampling and begins counting kernel submissions, only the next two kernels are registered. The GPU, however, computes all four kernels. As a result, NEON's measurement of GPU utilization is wrong, which can lead to unfairness. To overcome this problem, we do not only copy IB_PUT to the GPU during unshadow, but also fake resubmission for each kernel in the shadow channel by calling NEON's tracking callbacks manually.

The original version of NEON is able to enforce a limit to the number of kernels computed during the sampling phase. As an example, a scheduling policy may

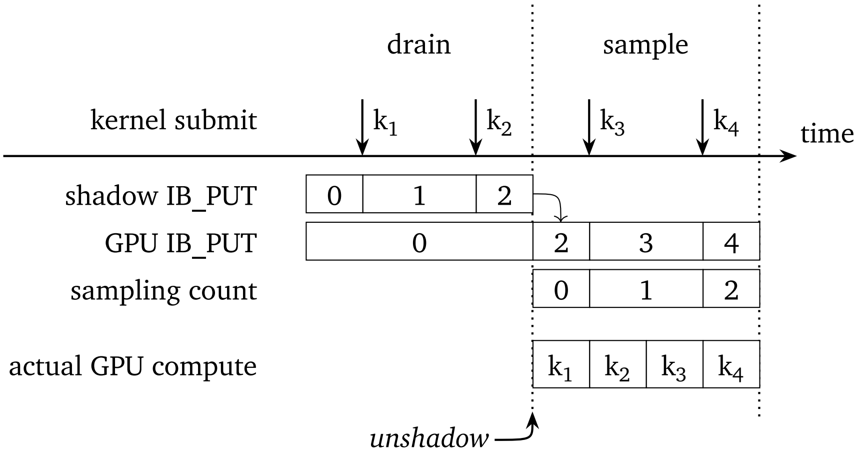


Figure 4.2: Wrong accounting after unshadow: NEON expects channels to be empty before sampling starts. With shadow channels, this is no longer true, because applications can submit kernels while being blocked. The GPU computes four kernels in total: Two, that were submitted into the shadow channel during the draining phase, and two submitted during sampling. This invalidates NEON's accounting, which only sees the later two kernels.

consider a particular number of registered kernels to be enough to infer the application's GPU occupation. When this number is reached, the sampling phase for the active application immediately ends and NEON blocks all further kernel submissions by the application. This mechanism is important to limit the duration of the sampling phase of each application. With shadow channels, however, applications may submit any number of kernels before sampling begins. Blindly copying IB_PUT to the GPU during unshadow allows all previously submitted kernels to run, which can easily exceed the designated duration of the sampling phase. To prevent such a situation, the policy can issue a special signal during kernel resubmission, which stops the unshadowing process, only increases IB_PUT on the GPU by the number of kernels permitted to execute by the respective scheduling policy and keeps the channel shadowed.

4.3.3 Grouping data structures

This section discusses our changes to NEON's internal data structures. These changes are required in order to allow separate accounting of tasks running inside virtual machines.

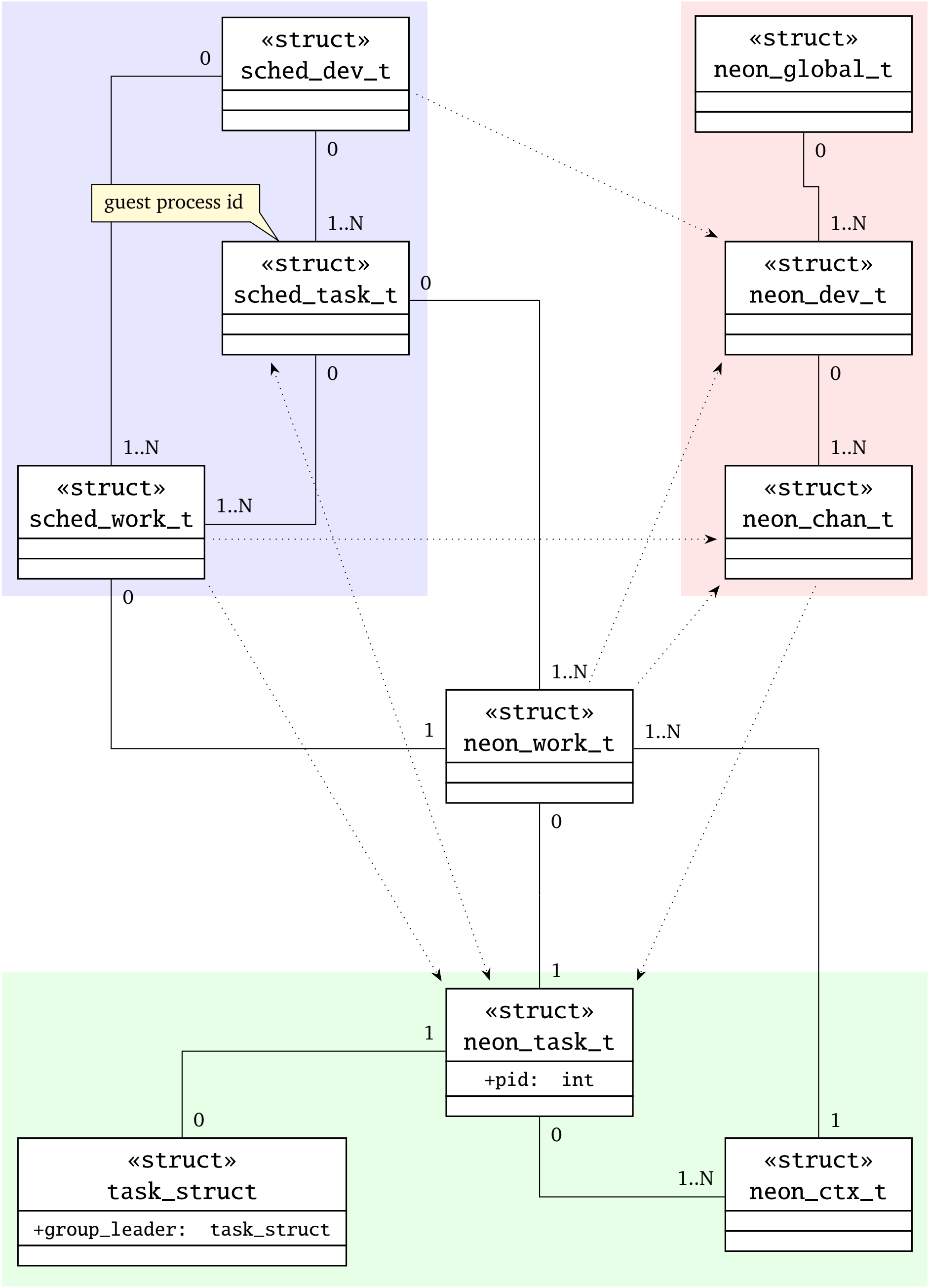


Figure 4.3: NEON's internal data structures. Straight lines denote direct references (pointers), dotted lines denote indirect references via ids. We add the guest process id to `neon_task_t` in order to support grouping. `task_struct` is a Linux kernel data structure. The colored groups are explained in Section 4.3.3.

Figure 4.3 depicts the core data structures of NEON and their relationships. The illustration is intended to be viewed similar to a *UML class diagram* [14]. Since NEON is written in C, which is not an object-oriented language and does not support the concept of classes, the illustration only loosely follows UML practices. The data structures can be divided into three groups: First, `neon_task_t` and `neon_ctx_t` abstract Linux kernel tasks and CUDA contexts, respectively. Structures in the second group represent different views on system hardware: The full system with multiple cards (`neon_global_t`), one GPU with multiple channels (`neon_dev_t`), and one channel (`neon_chan_t`). The third group contains all accounting-related data structures: `sched_dev_t` holds accounting data relevant for one GPU, `sched_task_t` accounts GPU usage of one task and `sched_work_t` abstracts channels from an accounting perspective. Finally, the central structure `neon_work_t` connects most data structures and represents an allocated channel that is in use.

Our prototype only supports one level of virtualization. This simplification allows us to keep most of NEON's internal data structures intact and introduce the required new functionality with little change: The most important data structure for accounting is `sched_task_t`, which represents a scheduling entity and holds values like the number of kernels submitted during sampling. Consequently, to handle more different scheduling entities, we create one `sched_task_t` per VM internal task. To keep scheduling entities distinguishable even if they share the same host task id, we add the guest process id, as depicted in Figure 4.3. Furthermore, we modify all instances where NEON accesses scheduling entities based on their host process id to also consider the guest process id.

4.3.4 Two level scheduling

In order to properly schedule tasks running inside potentially nested virtual machines, we assign a weight to each scheduling entity. An entity's weight represents the inverse of its target share of GPU time, compared to a task that runs in the hypervisor. In general, this weight depends on the number of children of each of the entity's ancestors. See Section 3.3.4 for details.

Since we only support one level of virtualization in our prototype, the weight calculation becomes easier: The weight of an entity equals the number of children its parent has. In other words, each entity's weight equals the number of its siblings, including itself. We update this number when a new task accesses the GPU for the first time and when a tasks exits.

NEON's DFQ scheduler attempts to balance GPU usage by measuring the average kernel runtime and the number of computed kernels of each application during the sampling phase. When an application overuses its fair share of GPU time, the scheduler blocks the application temporarily in order to allow other applications to run. See Section 2.4.6 for details. We modify NEON's scheduler to multiply the measured average kernel runtime of each application with its weight. As a result, NEON's scheduler considers applications with higher weights to overuse the GPU more often, which results in more frequent blocking of applications with higher weights.

4.4 Implementing nested scheduling

Our nested scheduling approach uses two different versions of NEON: One, that runs in the hypervisor and schedules between VMs, and one that runs and schedules inside the virtual machines. The hypervisor version is almost identical to the version from the paravirtual approach. Consequently, there is no implementation required besides adding compile-time flags to disable all paravirtual features except shadow channels. The version of NEON that runs inside the virtual machine, however, requires implementation work, because the virtual machine is a different environment than the host. More precisely, we perform the following modifications in order to run NEON inside a VM: First, we port NEON to compile against the newer Linux kernel that runs inside the virtual machine. Second, we add hooks to BLOGV's virtual GPU driver to extract the data which the original version of NEON reads from the NVIDIA driver. Last, we add support for BLOGV's virtual GPU.

The core of NEON consists of a loadable kernel module for the Linux kernel. The original version of NEON runs on Linux 3.4. BLOGV, however, uses the newer kernel version 3.14.29 for its virtual machines. We therefore port NEON's kernel module to compile against the newer kernel. This task is straight forward and only requires simple changes to the code that creates NEON's kernel thread.

NEON requires knowledge about channel creation requests by applications. Applications send ioctls to the GPU driver in order to issue such requests. The NVIDIA proprietary GPU driver contains a small open-source kernel module in order to comply with kernel license requirements. This module mostly forwards ioctls to the proprietary BLOB driver. NEON modifies this module and adds hooks in order to notice when a channel is allocated. In order to support virtual machines, we port these hooks to BLOGV's virtual GPU driver. Like NVIDIA's kernel module, which forwards ioctls to the proprietary driver, BLOGV's virtual GPU driver forwards calls to the hypervisor. Since the NVIDIA kernel module and BLOGV's driver work in a similar manner, adding NEON's hooks to BLOGV is straight forward.

NEON contains a list of reverse-engineered, hardcoded memory addresses for each supported device. These addresses point to memory regions in the device's *PCI BAR*, such as the the channel control register array. In order to support BLOGV's virtual PCI BAR device, we hardcode its memory addresses in NEON.

Since we want to share as much code as possible between the nested and the paravirtual approach, we also want to use shadow channels in order to prevent applications from accessing the GPU. However, when our implementation switches back from a shadow channel to the GPU channel, there is a small probability of triggering a severe *Machine Check Exception* (MCE) in the hypervisor. This exception causes the host to reboot immediately. The error description claims this MCE is caused by a hardware error. Because faulty software running in the virtual machine must not, under any circumstance, be able to stop the hypervisor, we believe this exception is not caused by our code. After investigating, we now believe this problem to be a known CPU errata, which Intel lists as HSD95. According to Intel's specification update document [21], this errata is triggered due to "certain internal conditions while running core and memory intensive operations". Since we cannot workaround this issue with such a vague description, we disable shadow channels for the nested scheduler and fall back to NEON's original blocking strategy, which holds applications in the page fault handler during kernel submission.

These modifications complete our proof of concept implementation. We use the following evaluation to demonstrate the basic feasibility of VM-internal accounting and scheduling. For a complete implementation, however, more work is required. This is especially true for NEON itself, which does not fully support SMP [27]. Adding more functionality and enabling SMP surfaces concurrency problems, some of which cannot be solved trivially by adding locks. Our primary approach contains workarounds for NEON's concurrency issues. Since the nested approach runs NEON multiple times, the chance of triggering such problems is much higher, which is why we recommend a partial rewrite of NEON for a full implementation. Furthermore, a complete solution must deal with the sampling interactions discussed in the design phase. For our prototype, we set the duration of both the internal sampling and freerun phases to 1 ms each, while keeping the duration of the hypervisors sampling phase at 5 ms, and freerun at 25 ms. These settings allow the internal scheduler to fit many complete cycles inside the hypervisors freerun phase, which reduces interaction effects to a minimum. This solution, however, is only possible for our simple prototype that does not need to support a larger number of tasks or nested virtualization.

5 Evaluation

This chapter covers the evaluation part of our work. The purpose of this evaluation is to verify we have reached our design goals defined in Section 3.1. In order to analyze the implementation of our paravirtual approach, we measure introduced overhead and verify that our system is able to enforce fairness in the host and in virtual machines. Furthermore, we demonstrate the feasibility of accounting and scheduling tasks inside virtual machines with our nested approach.

The remainder of this chapter starts with a description of our experimental setup in Section 5.1. Afterwards, we discuss how we generate predictable GPU load for the fairness tests in Section 5.2. We measure introduced overhead in Section 5.3, followed by an investigation on scheduling in Section 5.4.

5.1 Experimental setup

Our test system partially resembles the system used in the original NEON paper [28], because NEON depends on particular GPU driver versions and GPU models. We run all benchmarks and measurements on a system with an Intel Core i7-4770 processor, a NVIDIA GTX 480 GPU, and 16 GB of RAM. The host runs NEON's modified Linux kernel version 3.4.7 with KVM and QEMU 2.2.0 for the virtual machines. Our host system uses the NVIDIA proprietary GPU driver, version 331.62 with CUDA 6.0. The guests run Linux with kernel version 3.14.29 with BLoGV's virtual GPU driver and CUDA 6.0.

Since BLoGV currently does not support the CUDA Runtime API, we conduct our benchmarks with a CUDA Driver API based version [24] of the *rodinia* GPGPU benchmark suite [6]. Because NEON measures the runtime of multiple kernels before the first scheduling decision, we add a loop to the benchmarks that increases each benchmark's runtime into the range of full seconds. This loop repeats all data transfers and GPU computations. As a result, each benchmark issues a larger number of kernels, which allows scheduling by NEON. Furthermore, we modify the time measurement code in order to record kernel runtimes and duration of I/O operations separately.

For all our fairness measurements, we configure NEON to sample each application for 10 ms and set the duration of the freerun phase to 50 ms. We double the durations from the original NEON paper in order to account for the fact that kernels issued by some of our benchmarks have an average runtime of up to 4171 µs, much more than the longest average kernel runtime of 637 µs observed by the original authors of NEON [28]. Their sampling duration of 5 ms is too short to allow precise sampling of our longer running kernels, leading to suboptimal scheduling decisions.

5.2 Load generation

Similar to the evaluation in the original NEON paper, we use a *throttle* application for load generation in our fairness benchmarks. Throttle is a simple application that does nothing beside issuing kernels in an endless loop. These kernels occupy the GPU by busy waiting for a configurable amount of time. We repeat each of our fairness benchmarks multiple times with increasing configuration parameters in order to simulate applications with a range of different kernel runtimes.

We run our own version of throttle that uses the CUDA Driver API. Throttle also requires knowledge of the GPU's clock rate in order to properly wait for the configured amount of time. Since BLOGV currently does not support reading this value dynamically at runtime, we hardcode our GPU's clock rate.

5.3 Overhead

Adding new functionality to existing systems is rarely possible with zero additional overhead. Good modifications, however, keep new overhead to a minimum. Consequently, we decided that low overhead is a core goal of our implementation.

We only measure overhead introduced by our primary, paravirtual approach. The implementation of our nested approach only serves as a proof of concept for decentralized scheduling and does not feature a full design with a focus on low overhead.

In this evaluation, we conduct four major overhead measurements: We start with two measurements on overhead introduced by our GPU scheduling in Section 5.3.1. Afterwards, we investigate BLOGV's GPU virtualization as an additional source of overhead in Section 5.3.2. Finally, we measure the combined overhead in Section 5.3.3.

We perform overhead measurements as follows: All eight rodinia benchmarks run twice in different environments. As an example, the experiment on virtualization overhead runs all benchmarks once on the host, and once in the virtual machine. We record the results and compute the overhead as the slowdown from the first to the second run, expressed as a percentage.

A common problem when investigating overhead of scheduling systems is the interpretation of the results. It is often difficult to distinguish between slowdowns caused by actual overhead and slowdowns which are a result of scheduling itself: If a scheduling system grants less computation time to a benchmark, one might wrongly attribute the slowdown to increased overhead. In order to avoid such problems, we disable scheduling for the following overhead measurements. More precisely, we modify NEON to run as usual, but ignore the scheduling decision and always allow all applications to access the GPU in the freerun phase. This modification allows us to measure any overhead introduced by NEON's accounting mechanism without including scheduling effects.

5.3.1 Scheduling overhead

Our first measurement targets scheduling overhead. For this measurement, we perform both benchmarks runs in a virtual machine, one time without scheduling and one time with our paravirtual version of NEON. We therefore measure the effect of adding our scheduler to an existing setup of virtual machines. NEON performs all sampling normally, but never blocks applications during the sampling phase. We expect to measure a slowdown with enabled scheduling because of NEON's kernel submission interceptions during the sampling phase.

Kernel		**Kernel + I/O**	
Benchmark	Overhead	Benchmark	Overhead
BACKPROP	2.01 %	BACKPROP	2.71 %
BFS	4.06 %	BFS	1.27 %
HEARTWALL	18.97 %	HEARTWALL	2.17 %
HOTSPOT	-0.04 %	HOTSPOT	0.89 %
LUD	-0.03 %	LUD	0.91 %
NN	3.26 %	NN	1.13 %
SRAD_V1	4.00 %	SRAD_V1	6.17 %
SRAD_V2	2.47 %	SRAD_V2	2.08 %
Arithmetic mean	4.34 %	Arithmetic mean	2.17 %
Standard deviation	6.12 %	Standard deviation	1.75 %

Table 5.1: Overhead of paravirtual scheduling: Runtime increases of rodinia benchmarks after enabling paravirtual GPU scheduling in the virtual machine.

Table 5.1 presents the results. With arithmetic means of 4.34 % and 2.17 % for kernel and kernel + I/O runtimes, the general overhead is low and matches our expectations. The only notable exception is the measured overhead of 18.97 % in the kernel runtime of HEARTWALL. Note that this peak is only visible in the kernel runtime, but not in the measurement which includes I/O. This observation suggests overhead that only affects the submission or execution of the kernels, but not data transfers between system RAM and GPU memory.

We suspect NEON's kernel interceptions during the sampling phase to be a major source of all measured scheduling overhead. In order to account for guest tasks running in virtual machines, we modified NEON's interception code and introduced support for manipulating the EPT. We believe tracking applications by manipulating EPT entries to be more expensive than host page table manipulations, because EPT violations lead to a VM exit, which is more expensive than an OS entry on a page fault.

In our second measurement, we compare benchmark runtimes between the host and the virtual machine. In both cases, the paravirtual version of NEON runs in the hypervisor. Since the scheduler now samples the benchmarks in both runs, the major difference in scheduling is the employed tracking technique: NEON tracks host tasks by manipulating the normal page table, but uses EPT manipulations for tasks in the virtual machine.

Kernel		**Kernel + I/O**	
Benchmark	Overhead	Benchmark	Overhead
BACKPROP	1.70 %	BACKPROP	1.64 %
BFS	2.28 %	BFS	8.84 %
HEARTWALL	28.30 %	HEARTWALL	-0.99 %
HOTSPOT	-0.05 %	HOTSPOT	1.06 %
LUD	-0.04 %	LUD	1.10 %
NN	1.24 %	NN	-35.77 %
SRAD_V1	2.17 %	SRAD_V1	4.11 %
SRAD_V2	0.63 %	SRAD_V2	1.94 %
Arithmetic mean	4.53 %	Arithmetic mean	-2.26 %
Standard deviation	9.65 %	Standard deviation	13.85 %

Table 5.2: Cost of virtualized tracking: Runtime increases of rodinia benchmarks when comparing execution on the host with execution on the guest. NEON samples, but does not schedule in both cases.

Table 5.2 presents the results. Again, the overhead in general is low, with an average slowdown for kernel runtimes of 4.53 %. The slowdown in the kernel runtime of the HEARTWALL benchmark grows to 28.30 % in this setup. Additionally, the total runtime of the NN benchmark decreases by 35.77 %, which is unexpected.

These results cannot be explained with increased costs from virtualized tracking alone. The main problem are the outliers HEARTWALL and NN. Overhead from increased tracking should affect all applications. It certainly cannot be the cause for NN's even faster runtime in the virtual machine compared to the host.

We therefore suspect an additional source of overhead to influence this measurement: Overhead from BLoGV's GPU virtualization. In order to separate the two, we first conduct a microbenchmark on the increased costs of kernel submissions, and measure pure virtualization overhead afterwards.

5.3.1.1 Interception of kernel submissions

Our paravirtual version of NEON is able to account the GPU usage of single applications in virtual machines. The employed channel access tracking technique resembles the one used in the original version of NEON, but uses VM exits, which are presumable more expensive than normal OS entries. We measure the cost of both tracking techniques with a microbenchmark in order to quantify the difference.

For this measurement, we use a simple CUDA application that only submits empty kernels in an endless loop, and measure how long the API call `cuLaunchKernel` takes. In order to submit the kernel to the GPU, the CUDA API increments a pointer in a channel control register. With both tracking techniques, this causes exactly two traps: In the non-virtualized case a page fault, followed by a kernel entry triggered by the trap flag. In the virtualized case an EPT violation, followed by a hypervisor entry caused by the monitor trap flag. We expect VMM entries to be more expensive than OS entries, resulting in longer durations of kernel submissions during sampling in the virtual machine. During freerun, however, we expect the durations to be similar.

Figure 5.1 shows an excerpt of the results. The total measurement runs for 6 s, shown are the last 200 ms of the recording to allow for cache warmup. For this benchmark, we use a longer sampling phase of 20 ms and only 60 ms of freerun. The phases are clearly visible in both graphs: Kernel submissions take longer during sampling in both cases. Note that the graphs come from independent measurements, the phases are therefore not synchronized.

The increased duration of kernel submissions in the virtual machine matches our expectations, even though the difference is much larger than we expected. A kernel

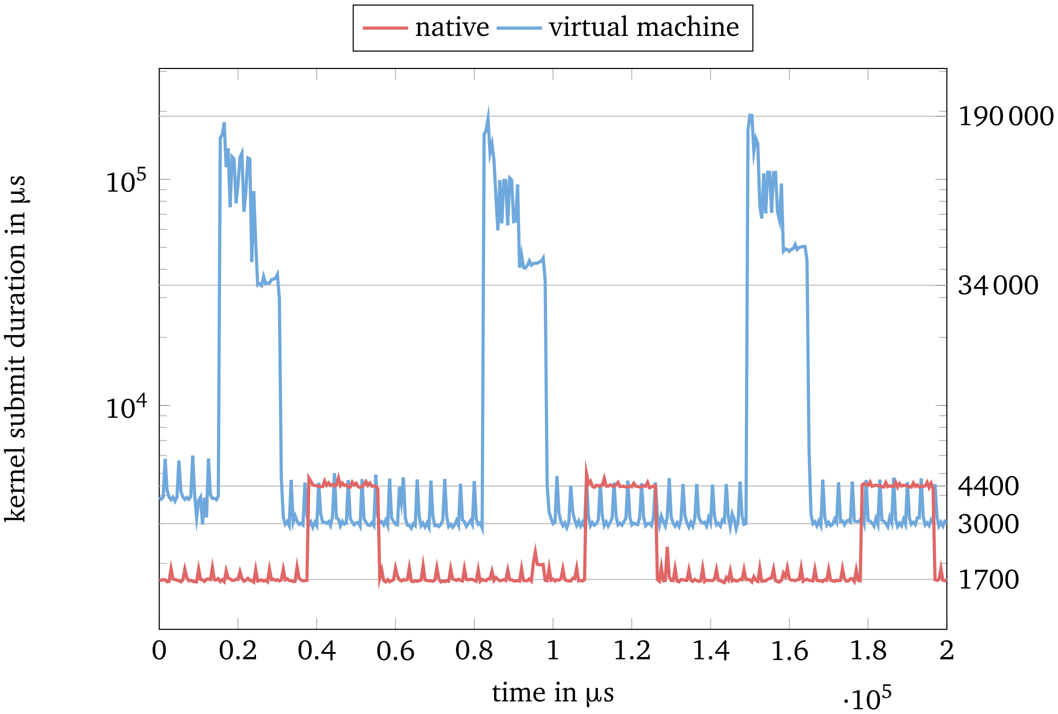


Figure 5.1: Microbenchmark: Durations of kernel submissions on the host and in virtual machines. The plot shows the duration of kernel submissions in an endless loop with a CUDA sync after each iteration. NEON is set to 20 ms sampling and 60 ms freerun. No actual scheduling takes place. A kernel submission during sampling takes approximately 4400 µs on the host and 34 000 µs to 190 000 µs in the virtual machine with the paravirtual approach.

submission during sampling takes about 4400 µs on the host and approximately 34 000 µs to 190 000 µs in the virtual machine. Even in the best case, virtualized tracking is therefore almost one order of magnitude slower than native tracking.

The durations of kernel submissions during freerun, however, are surprising. A kernel submission during freerun takes about 1700 µs on the host and about 3000 µs in the virtual machine. We expected the difference to be smaller. Our investigation shows that the CUDA library performs more actions on `cuLaunchKernel` than expected. As an example, an additional ioctl is issued for each kernel submission. Note that this additional ioctl is not the source of the increased duration in the virtual machine, since BLOGV contains an optimization and ignores it. Ultimately, we were unable to find the cause for the longer kernel submission duration during freerun in the virtual machine. We suspect the CUDA library to perform some additional task, which behaves differently in the virtual machine.

Note that, even though the overhead measured in this microbenchmark is higher than expected, the overall impact is limited, since NEON only tracks kernel submissions during the sampling phase and the submission itself is only a small part of the total benchmark runtime. The results in the previous measurement in Table 5.2 confirm that the slower sampling only leads to a very limited application overhead in most benchmarks.

In any case, this measurements proves that virtualized tracking is more expensive than native tracking on the host. While this does explain the slight runtime increases observed in most benchmarks, virtualized tracking is likely neither the cause for the slowdown observed in the HEARTWALL benchmark, nor the cause for the speedup in the NN benchmark.

5.3.2 Virtualization overhead

This measurement targets overhead introduced by BLOGV's GPU virtualization. Many GPU calls take longer in the virtual machine, because BLOGV's virtual GPU driver forwards them to the hypervisor. Overhead from BLOGV's GPU call forwarding is not caused by our scheduling approach, but influences all measurements that use virtual machines. We therefore conduct a measurement that compares benchmark runtimes with and without virtualization in order to isolate this effect.

Kernel		**Kernel + I/O**	
Benchmark	Overhead	Benchmark	Overhead
BACKPROP	0.81 %	BACKPROP	2.82 %
BFS	0.10 %	BFS	4.76 %
HEARTWALL	12.68 %	HEARTWALL	-0.78 %
HOTSPOT	0.14 %	HOTSPOT	-0.96 %
LUD	-0.03 %	LUD	0.61 %
NN	0.03 %	NN	-32.68 %
SRAD_V1	0.09 %	SRAD_V1	0.39 %
SRAD_V2	0.04 %	SRAD_V2	1.45 %
Arithmetic mean	1.73 %	Arithmetic mean	-2.81 %
Standard deviation	4.43 %	Standard deviation	12.19 %

Table 5.3: BLOGV's virtualization overhead: Runtime increases of rodinia benchmarks when comparing execution on the host with execution on the guest. No accounting or scheduling is involved.

Table 5.3 presents the results. For this benchmark, we compare the runtimes of all eight rodinia benchmarks on the host with the runtimes in BLoGV's virtual machine. No scheduling or accounting is involved in any case. While the results show that BLoGV only introduces negligible overhead in most benchmarks, the results again include the same anomalies in the HEARTWALL and NN runtimes we already observed in previous measurements.

We therefore conclude these overheads are caused by BLoGV's GPU virtualization. One possible explanation for the increased runtime of HEARTWALL is the fact that HEARTWALL is heavily I/O bound and has very small actual kernel runtimes. We measure a kernel execution duration to total benchmark runtime ratio of approximately 1:100. The HEARTWALL kernels only run for a accumulated duration of approximately 25 ms. The kernels of all other benchmarks, by contrast, have an accumulated kernel runtime of at least one second each. We therefore suspect initial costs, for example from cold caches, to be the source of the measured overhead. With such a short runtime, a large share of the measured absolute kernel runtime consists of the initial costs.

The other outlier is the kernel + I/O runtime of the NN benchmark. The runtime of NN shrank by over 35 % as a result of running NN in the virtual machine. We ultimately decided against investigating this further, since it is evidently caused by BLoGV and not by our scheduler.

5.3.3 Combined overhead

The final overhead measurement includes all possible sources of overhead in order to verify our paravirtual approach is practical. For this benchmark, we run rodinia one time on the host without any scheduling, and one time in the guest with our paravirtual version of NEON. We expect this setup to show the highest levels of overhead, because it includes all costs from virtualization and scheduling.

Table 5.4 presents the results. Again, HEARTWALL and NN differ from the rest of the results. We expected this behavior, since it already occurred in the previous measurements. Note that our measurements show that both outliers primarily originate from BLoGV and are not introduced by our scheduler.

Except for HEARTWALL and NN, the combined overhead is still reasonably low. The negative outlier from the NN benchmark causes the average overhead of the total benchmark runtimes to be negative, which limits the meaningfulness of the average for the other benchmarks. We therefore compute the arithmetic mean again without the negative outlier from NN and arrive at a total average overhead for both GPU scheduling and GPU virtualization of 3.94 %. With this result, we think it is safe to say that we have reached our goal of minimizing overhead.

Kernel		Kernel + I/O	
Benchmark	Overhead	Benchmark	Overhead
BACKPROP	0.66 %	BACKPROP	1.87 %
BFS	4.02 %	BFS	11.56 %
HEARTWALL	28.81 %	HEARTWALL	0.14 %
HOTSPOT	-0.05 %	HOTSPOT	1.62 %
LUD	-0.03 %	LUD	1.59 %
NN	2.82 %	NN	-35.98 %
SRAD_V1	4.28 %	SRAD_V1	6.94 %
SRAD_V2	2.56 %	SRAD_V2	3.83 %
Arithmetic mean	5.38 %	Arithmetic mean	-1.05 %
Standard deviation	9.62 %	Standard deviation	14.59 %

Table 5.4: Total cost of GPU virtualization and GPU scheduling: Runtime increases of rodinia benchmarks when comparing execution on the host without scheduling to execution on the guest with our paravirtual version of NEON.

5.4 Scheduling

Our design goal is to create fairness among applications competing for the GPU in each layer of virtualization. Since our prototype is limited to the host plus one layer of virtualization, we can easily verify both in separate experiments.

We first test the original version of NEON without virtualization to establish a baseline in Section 5.4.1. Afterwards, we evaluate our paravirtual approach in Section 5.4.2, followed by a brief evaluation of our nested proof of concept in Section 5.4.3.

5.4.1 Baseline

Our implementations use NEON as a basis. In order to understand later benchmark results, we first establish a baseline of NEON's scheduling behavior without virtualization.

For the following tests, we run each of the rodinia benchmarks together with the throttle application. We run each benchmark ten times with increasing throttle kernel runtimes. Without any scheduling, this setup causes the GPU's internal round-robin scheduler to grant increasingly larger shares of GPU time to throttle, which results in longer benchmark runtimes. With a perfect scheduler, however, the benchmark runtime stays constant, independent of the runtime of the throttle kernels, because both the benchmark and throttle each get exactly 50 % of the available GPU computation time.

Figures 5.2 to 5.4 present the results. Without external scheduling, the effects of the GPU's internal round-robin scheduler are clearly visible: The benchmark runtime increases linearly with the throttle kernel runtime. After we enable NEON, this effect is mostly gone and the runtime only slightly increases with larger throttle kernels for most benchmarks. Still, this is a limitation of NEON, a perfect scheduler keeps the runtime constant without any increase. Since our approach does not alter the core operation principle of NEON, we take note of this behavior and expect our approaches to behave similar in their evaluation.

Note that in all these experiments, the quality of the achieved fairness depends on the response of the benchmark runtimes to the increasing throttle kernel runtimes, and not on their initial absolute values. As an example, comparing HOTSPOT and NN may lead to the assumption that NEON's scheduling works well for HOTSPOT and creates large amounts of overhead for NN. This conclusion, however, is wrong. The much longer runtime of NN with NEON is in fact the desired behavior of any fairness-enforcing GPU scheduler: NEON measures an average kernel runtime of 4171 µs for NN's kernels. The throttle kernels, however, only run for 100 µs in the first measurement. Since GPU computation time shares linearly depend on kernel runtimes in the absence of scheduling, NN grabs about 97,6 % of the available GPU time. NEON correctly identifies this problem and enforces a fair 50 % share, which roughly doubles the runtime of NN.

In general, the results show that NEON achieves fairness in all benchmarks. In some benchmarks, however, the runtimes with NEON vary more than in others. One example is SRAD_V2, which shows non-linear variations in the benchmark runtimes. Again, a look at the average kernel runtimes helps with understanding this phenomenon. NEON measures an average kernel runtime of 2699 µs for the SRAD_V2 benchmark. Since NEON's sampling phase is 10 ms long, only a very small number of kernel runs contribute to NEON's measurements. As a result, the average is less precise. Since NEON's scheduling decisions depend on this average, slightly off measurements result in mild unfairness, as observed in the benchmark.

5.4.2 Paravirtual approach

After establishing a baseline for NEON's scheduling capabilities, we continue with our paravirtual approach. Here, we perform two different experiments: First, we repeat the previous benchmark in the virtual machine in Section 5.4.2.1. Second, we test a new scenario for fairness in the host in Section 5.4.2.2.

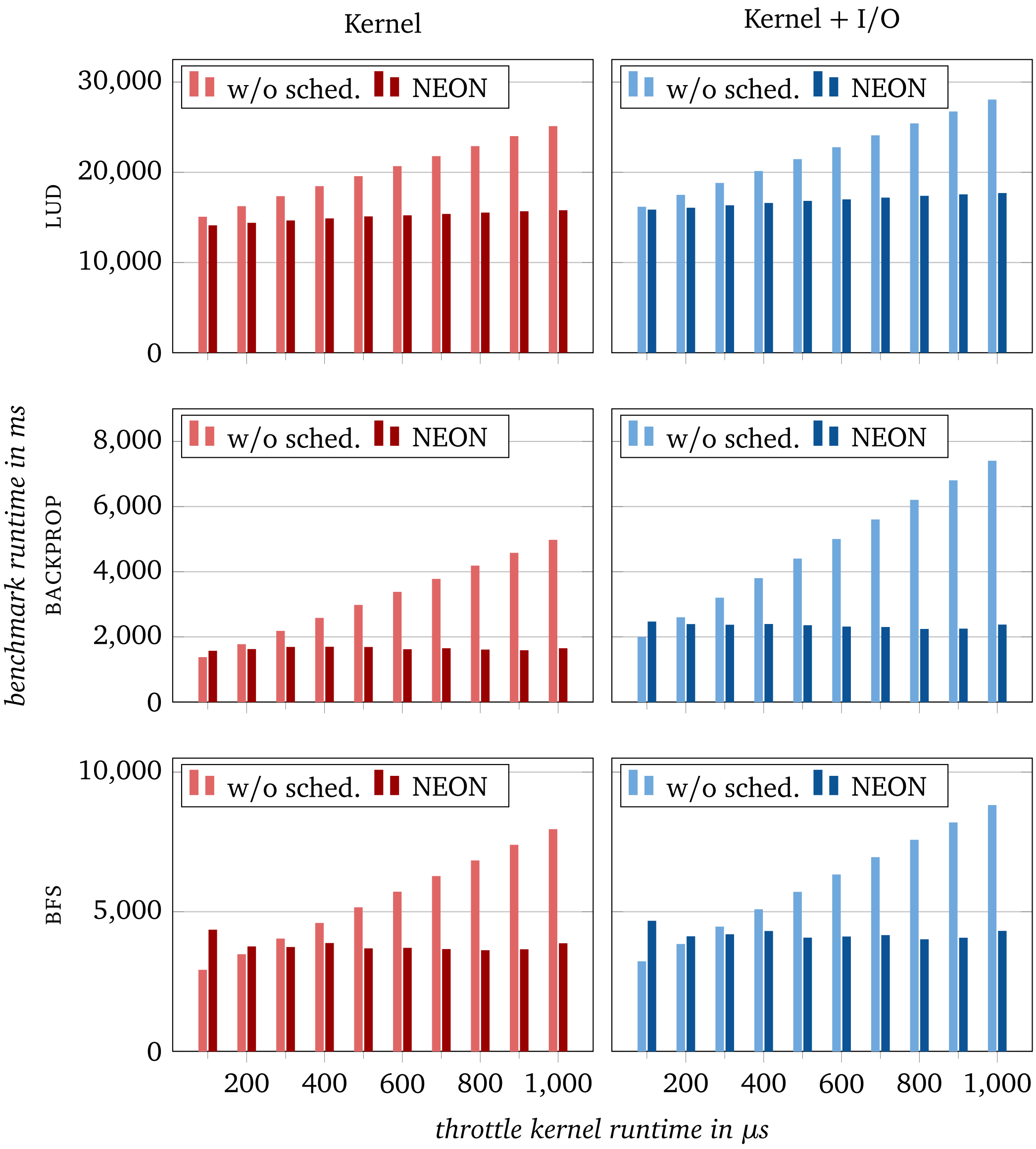


Figure 5.2: Baseline of NEON's scheduling capabilities. Shown are the runtimes of the rodinia benchmarks LUD, BACKPROP, and BFS. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

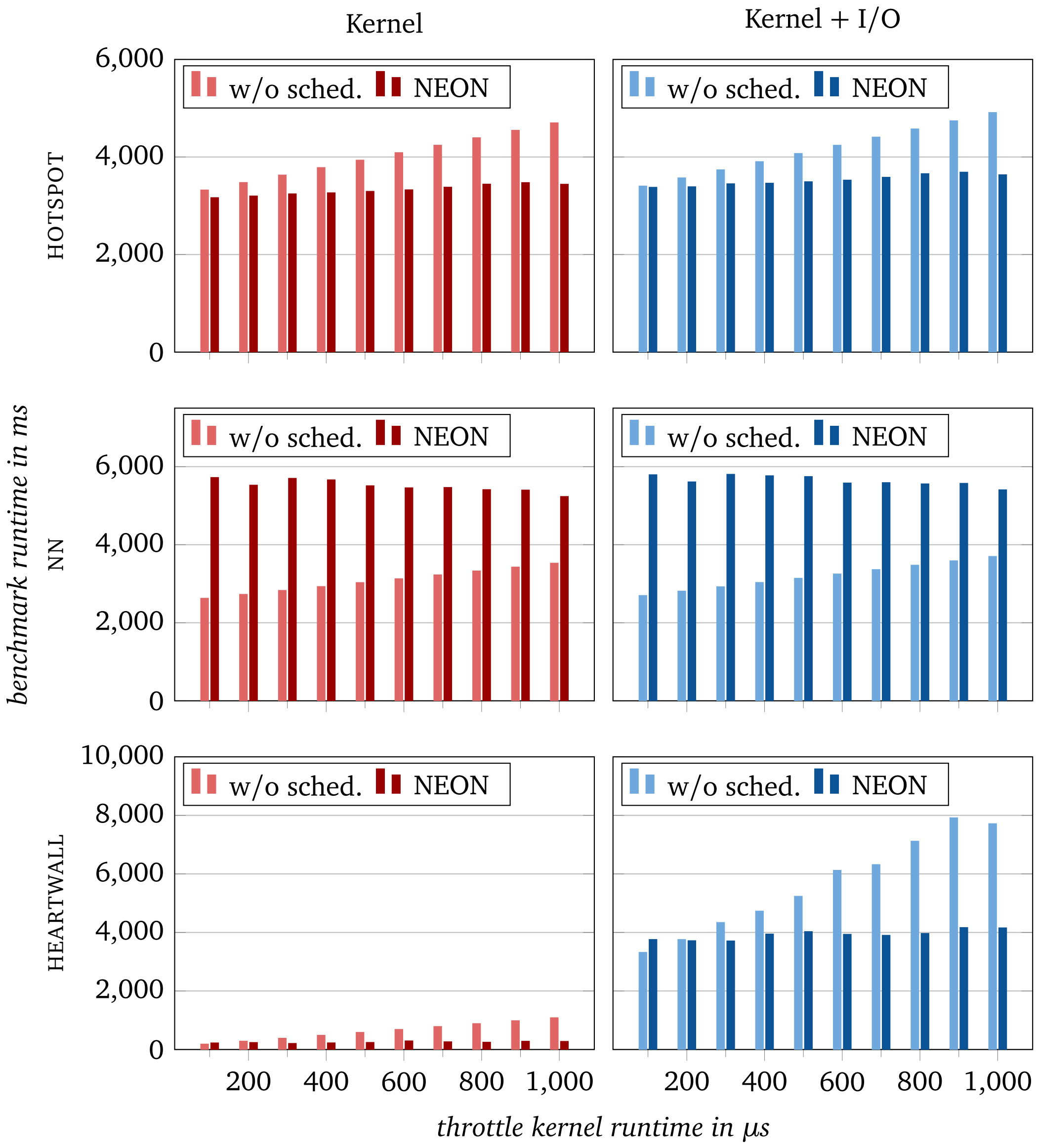


Figure 5.3: Baseline of NEON's scheduling capabilities. Shown are the runtimes of the rodinia benchmarks HOTSPOT, NN, and HEARTWALL. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

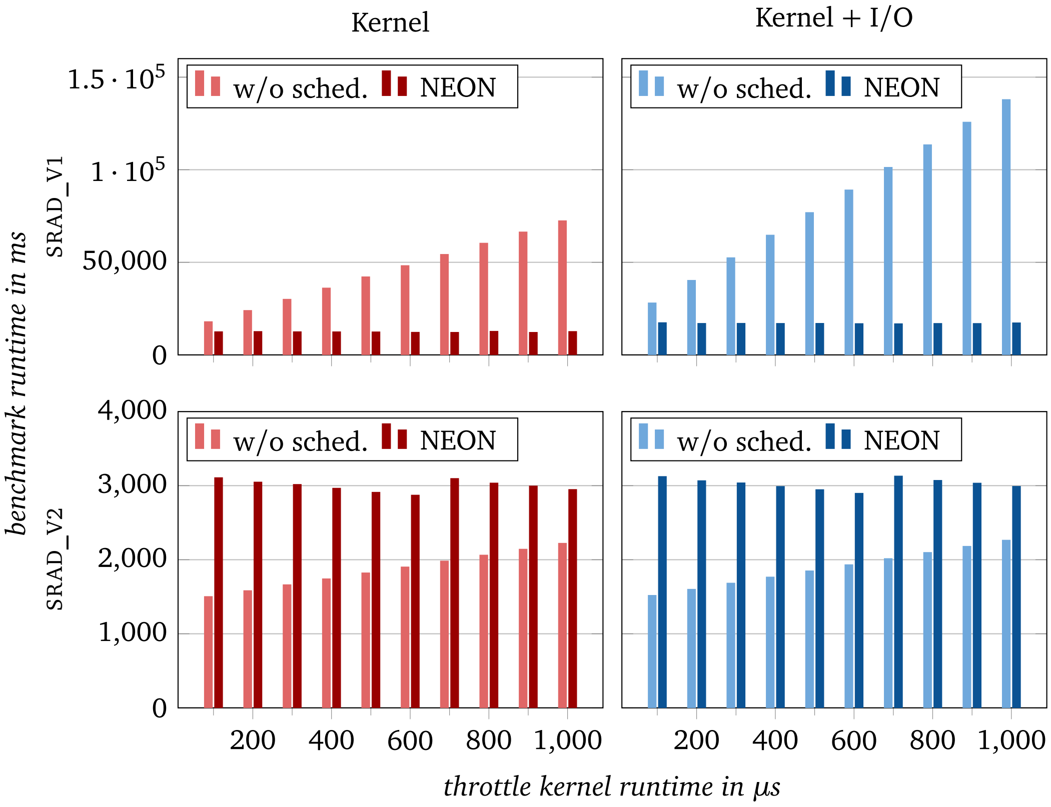


Figure 5.4: Baseline of NEON's scheduling capabilities. Shown are the runtimes of the rodinia benchmarks SRAD_V1 and SRAD_V2. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

5.4.2.1 Fairness inside virtual machines

With this experiment, we demonstrate our schedulers ability to schedule tasks inside the virtual machine from the hypervisor. Except for the execution in the virtual machine, the setup is the same as in the baseline experiment in Section 5.4.1. We run each rodinia benchmark together with throttle in the same VM, and repeat each measurement with increasing throttle kernel runtimes.

Figures 5.5 to 5.7 present the results, which look almost exactly like the results of the baseline experiment. Again, the runtimes increase linearly in the absence of scheduling. Our paravirtual version of NEON correctly identifies the situation and enforces fairness. Like in the baseline measurement, the runtime of some benchmarks still increases slightly with larger throttle kernel runtimes. Our scheduler therefore achieves very good, but not perfect scheduling results. This behavior matches the results of the baseline experiment which does not use virtualization. We therefore conclude that our paravirtual version of NEON successfully applies NEON's scheduling capabilities to tasks running inside virtual machines.

5.4.2.2 Fairness between virtual machines

In the previous section, we demonstrated our scheduler's ability to enforce fairness among competing tasks inside a virtual machine. The purpose of the experiment in this section is to demonstrate our scheduler's ability to enforce fairness between entire virtual machines in the host.

For this test, we run the rodinia benchmarks on the host. Additionally, we run an increasing number of throttle applications inside one virtual machine. We run up to eight throttle instances, since this is the maximum number of applications currently supported by BLoGV. Together with the benchmark, the GPU therefore sees up to nine different applications. A scheduler that allocates the same share of GPU time to each application violates the host's fairness requirements, since the virtual machine as a whole and the benchmark get different shares of GPU time. We expect our version of NEON to detect this situation, treat all throttle instances in the virtual machine as one and grant the benchmark the same runtime as the throttles combined. In other words, increasing the number of throttle instances must not impact the benchmark runtime. For this experiment, we configure QEMU to emulate eight virtual CPUs for the virtual machine, in order to guarantee parallel execution of all eight throttle instances.

Figures 5.8 to 5.10 present the results. Again, the runtimes without scheduling grow linear, which confirms our expectations and demonstrates the inherent unfairness without external scheduling. In most benchmarks, however, the linear growth

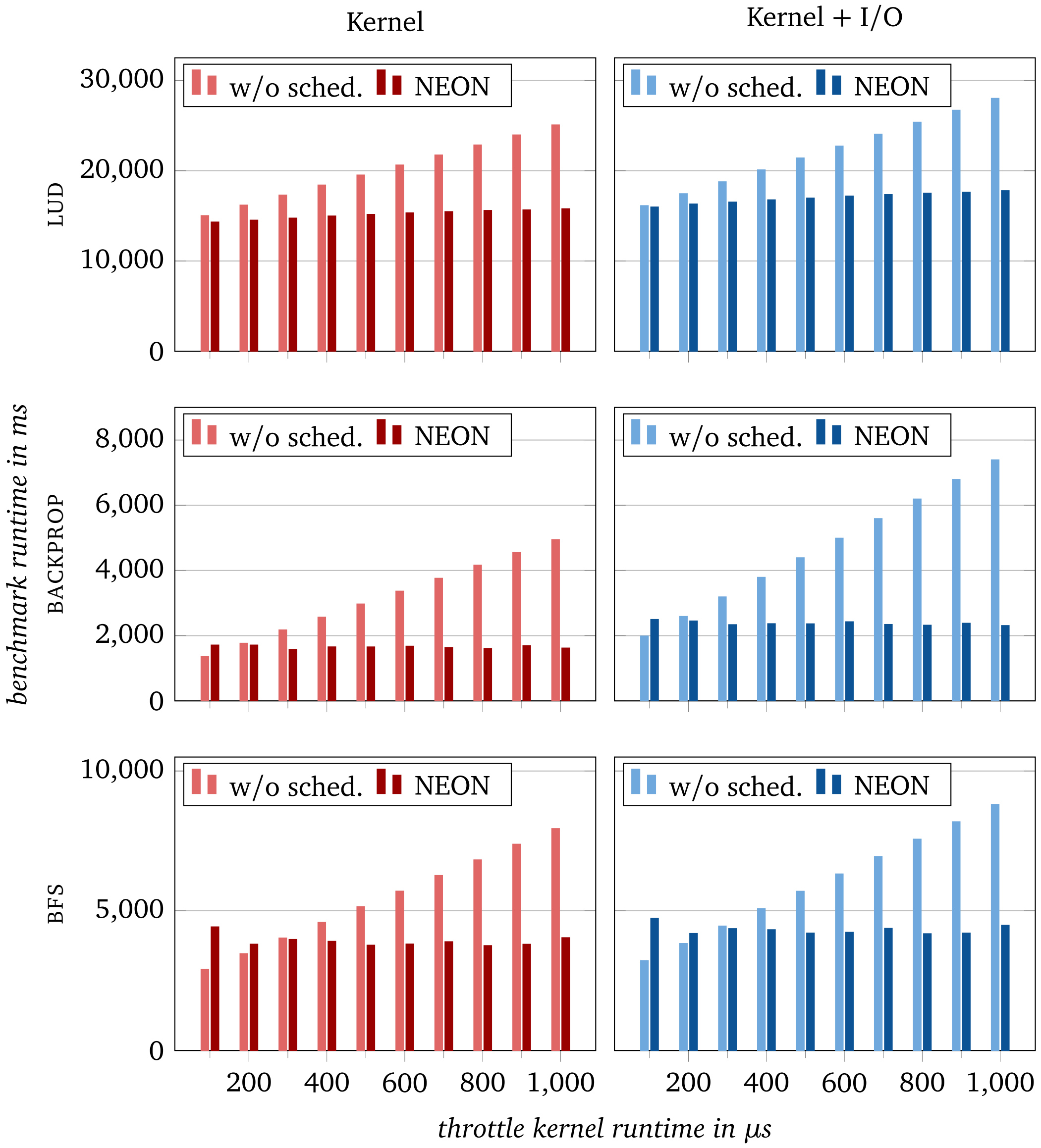


Figure 5.5: Fairness enforcement inside a virtual machine with our paravirtual approach. Shown are the runtimes of the rodinia benchmarks LUD, BACKPROP, and BFS. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

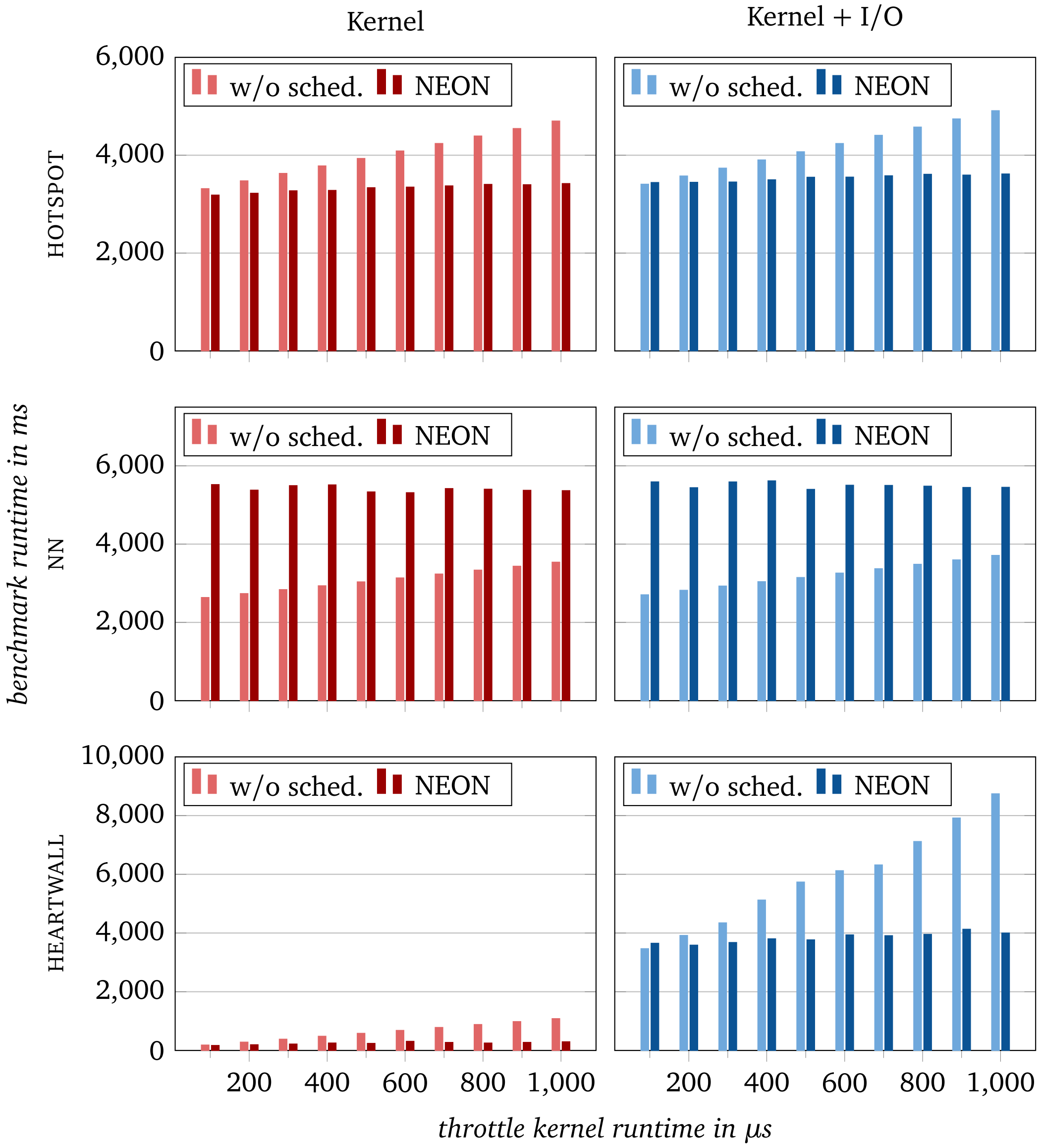


Figure 5.6: Fairness enforcement inside a virtual machine with our paravirtual approach. Shown are the runtimes of the rodinia benchmarks HOTSPOT, NN, and HEARTWALL. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

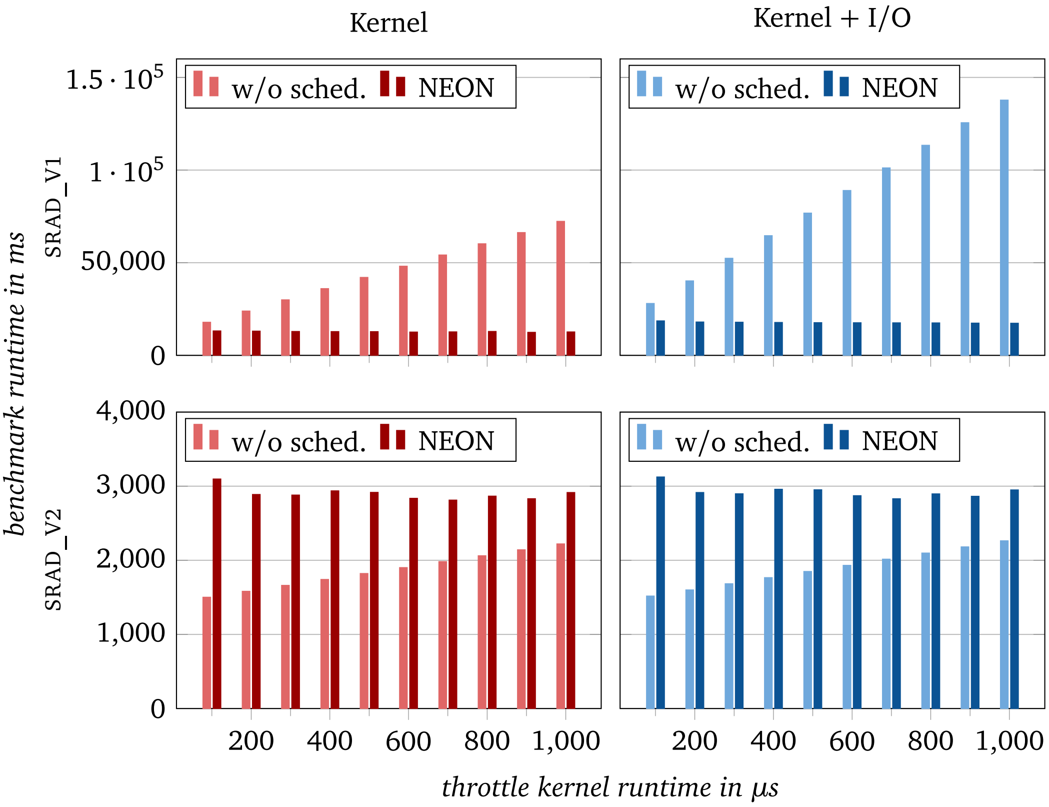


Figure 5.7: Fairness enforcement inside a virtual machine with our paravirtual approach. Shown are the runtimes of the rodinia benchmarks SRAD_V1 and SRAD_V2. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

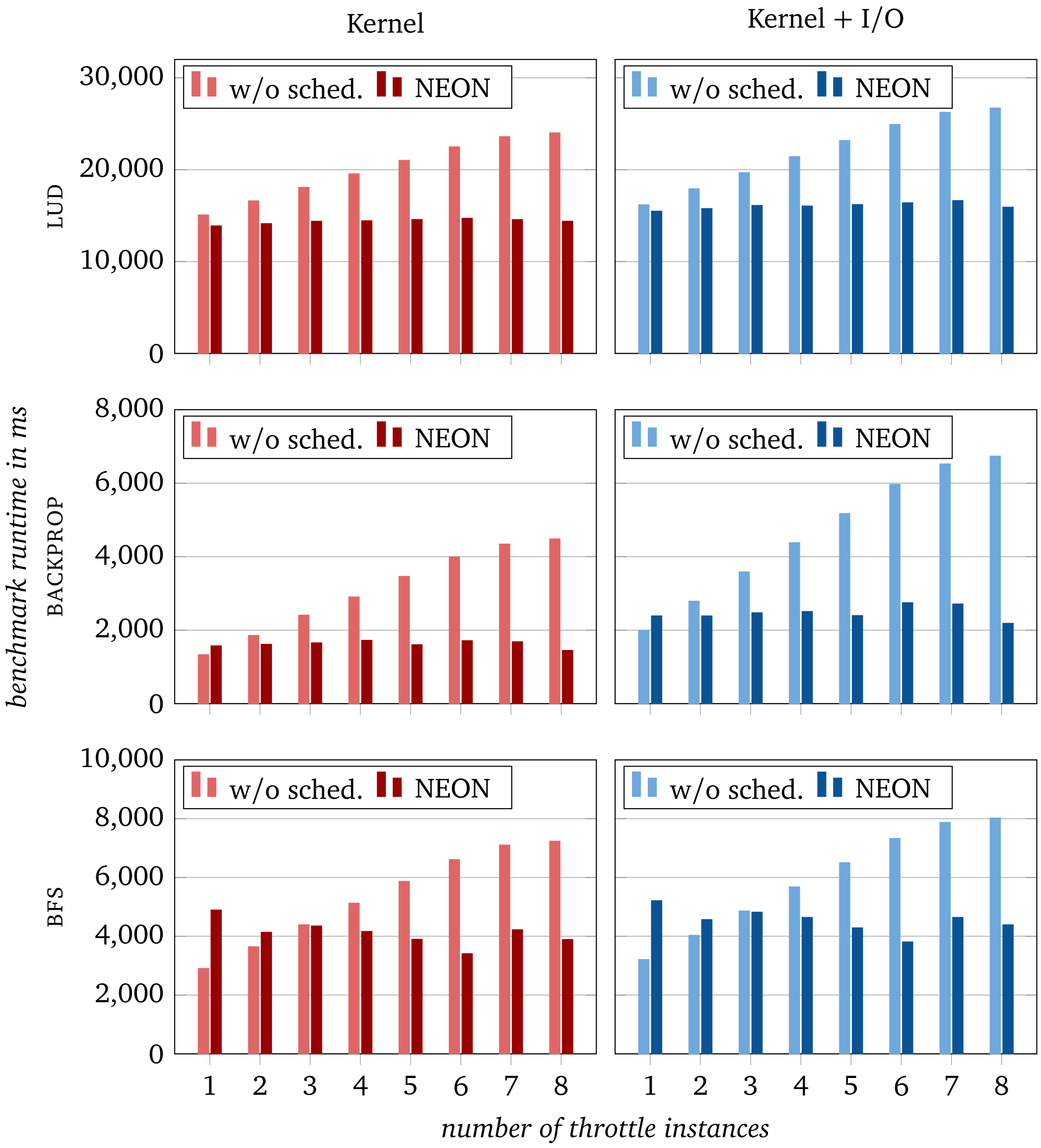


Figure 5.8: Fairness enforcement between a virtual machine and a host task with our paravirtual approach. Shown are the runtimes of the rodinia benchmarks LUD, BACKPROP, and BFS. Each benchmark runs eight times on the host, with increasing numbers of throttle instances in the guest, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of the number of throttle instances in the virtual machine.

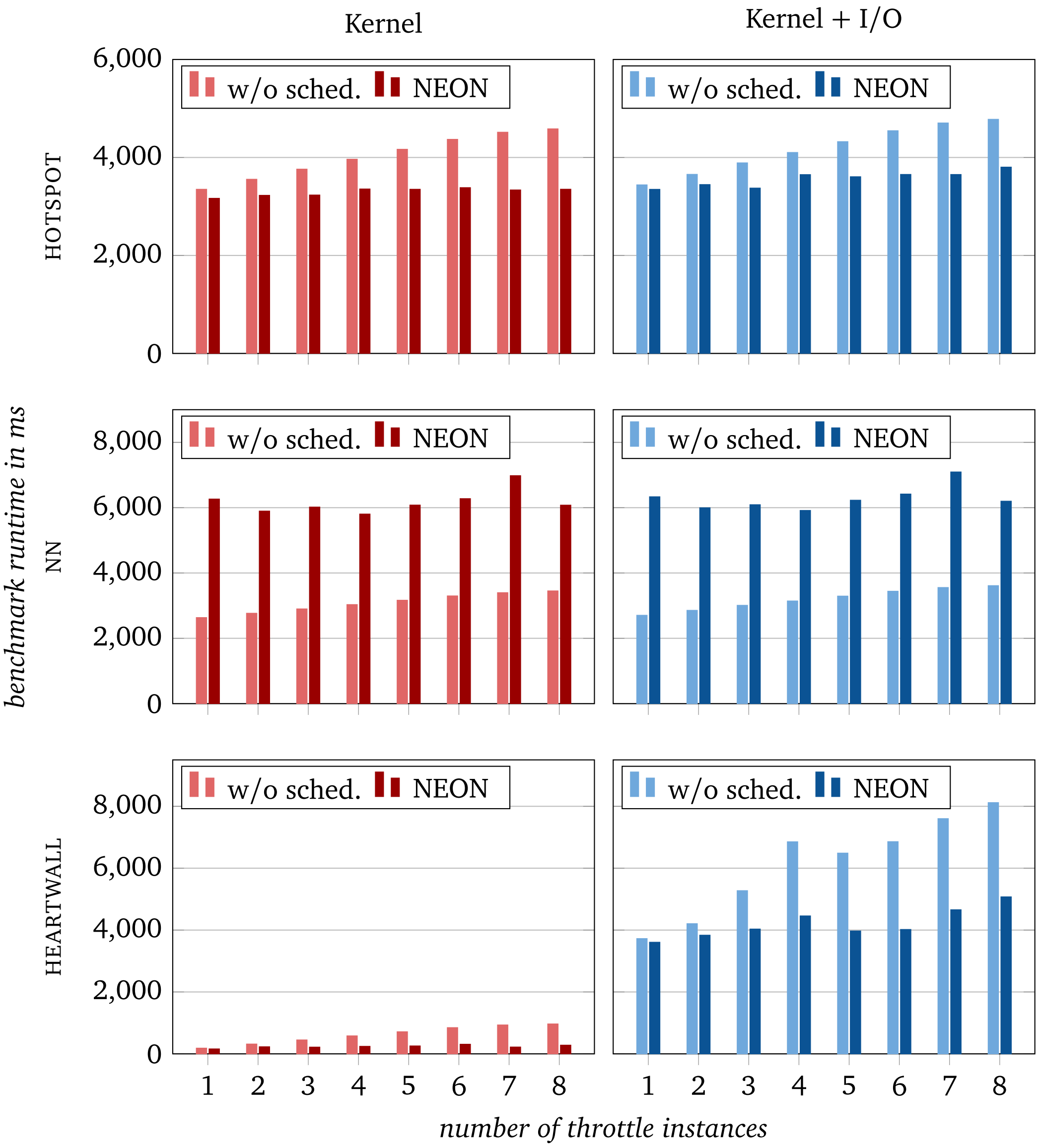


Figure 5.9: Fairness enforcement between a virtual machine and a host task with our paravirtual approach. Shown are the runtimes of the rodinia benchmarks HOTSPOT, NN, and HEARTWALL. Each benchmark runs eight times on the host, with increasing numbers of throttle instances in the guest, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of the number of throttle instances in the virtual machine.

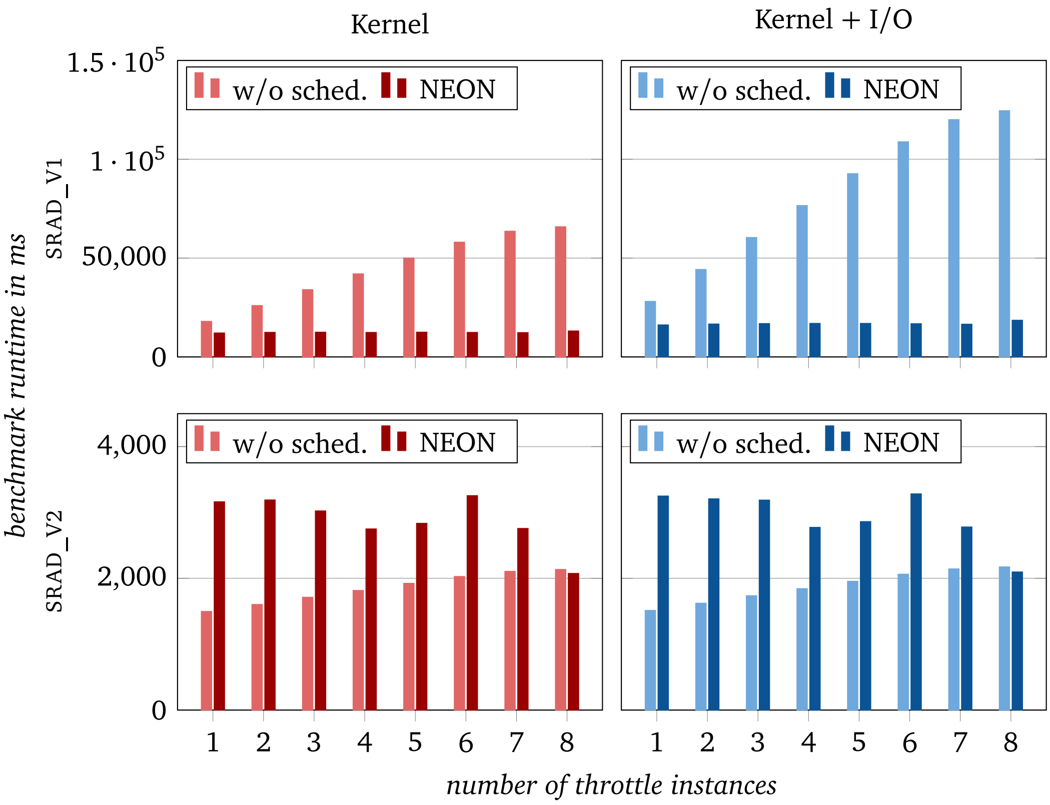


Figure 5.10: Fairness enforcement between a virtual machine and a host task with our paravirtual approach. Shown are the runtimes of the rodinia benchmarks SRAD_V1 and SRAD_V2. Each benchmark runs eight times on the host, with increasing numbers of throttle instances in the guest, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of the number of throttle instances in the virtual machine.

does not extend to the last measurement with eight throttle instances. The reason for this are hardware limitations of our test machine, which features a CPU with a total of eight hardware threads. In the last measurement, however, we run eight throttle instances plus the benchmark, which exceeds the hardware capabilities, limits concurrency, and therefore reduces the slowdown effect of adding more throttle instances.

In general, our paravirtual version of NEON is able to limit the cumulative GPU usage of the throttle instances and thus enforces fair GPU sharing in the host. Even though the values vary more than in the previous experiment on VM internal fairness, and our scheduler shows a slight tendency to overcompensate with the maximum number of throttles, we think it is again fair to say we also have reached our goal of enforcing fairness between competing applications in all virtualization layers.

5.4.3 Nested approach

In our final experiment, we demonstrate the basic feasibility of nested scheduling, which performs both sampling and scheduling inside the virtual machine. The evaluation setup is the same as in the experiment for VM-internal fairness in the paravirtual approach: The rodinia benchmarks run inside the virtual machine together with the throttle application, which we configure to send increasingly long running kernels. The only difference is the scheduler: In this experiment, the host scheduler stays disabled, and our nested version of NEON runs inside the virtual machine.

A feature complete version of the nested approach would still execute a scheduler in the host in order to enforce fairness between the virtual machines and host tasks. Since our prototype only serves as a proof of concept, however, this scenario is currently unsupported. Still, we configure the nested scheduler to use very short sampling and freerun phases of 1 ms each. As discussed in section 3.4.1, this reduces sampling interactions which affect the measurement results in a negative way. Even though we do not run an external scheduler and no such effects can appear in this experiment, we still configure short phases in order to demonstrate that NEON works with these settings.

Figure 5.11 shows the results of three selected benchmarks. We omitted the other benchmark results due to unresolved concurrency issues in NEON, which we discussed in Section 4.4.

Again, the benchmark runtime increases linearly without scheduling, indicating unfairness. Our nested scheduling prototype detects this unfairness and corrects it

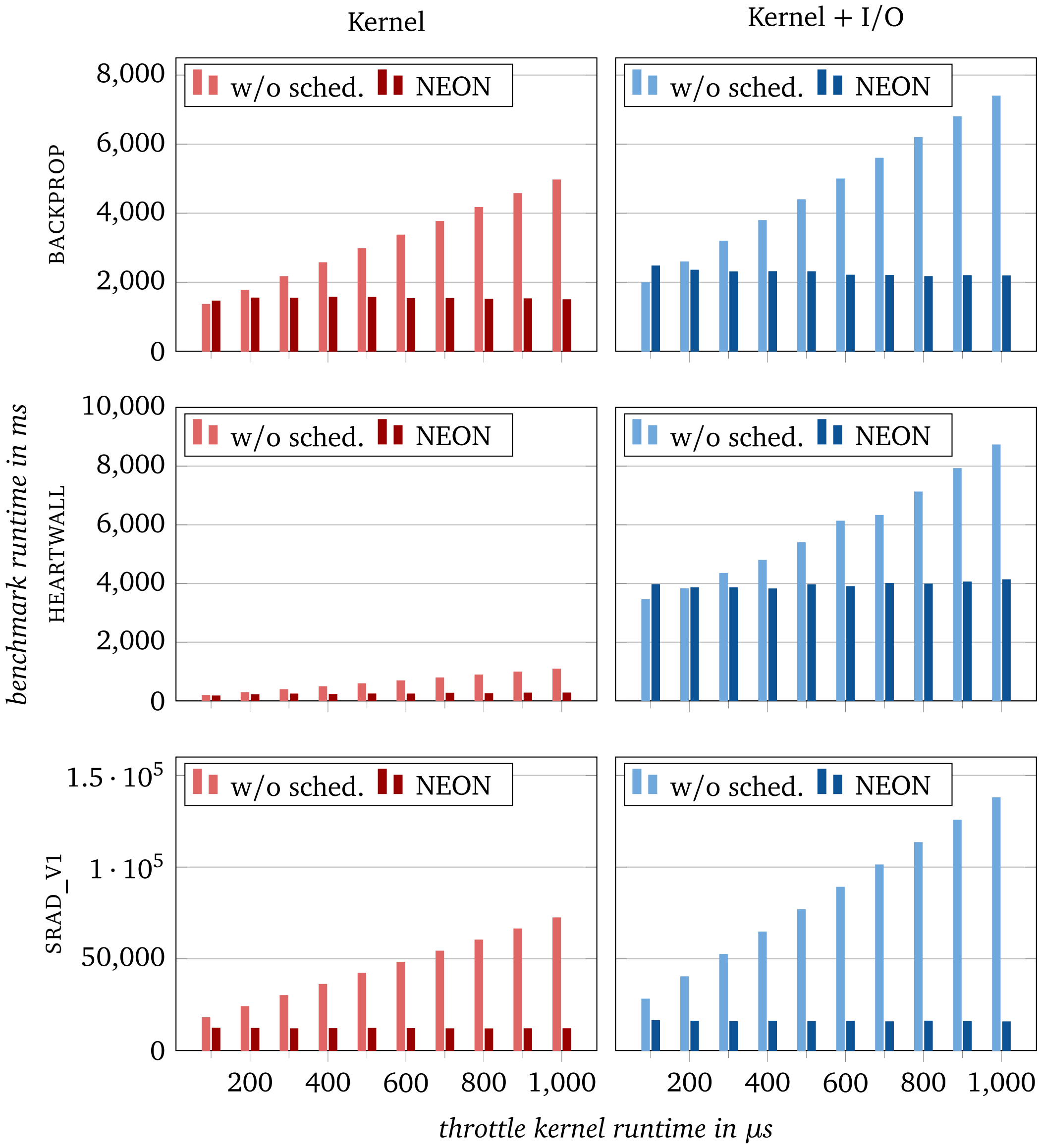


Figure 5.11: Fairness enforcement inside a virtual machine with our nested approach. Shown are the runtimes of the rodinia benchmarks BACKPROP, HEARTWALL, and SRAD_V1. Each benchmark runs ten times alongside the throttle application. We configure throttle to create increasing amounts of load. Without any scheduling, throttle occupies increasingly larger shares of GPU time, resulting in longer benchmark runtimes. A perfect scheduler keeps the benchmark runtimes constant, independent of throttle's kernel runtime.

by halting the throttle instances temporarily in order to grant the benchmark more computation time. The results clearly show that our prototype works and therefore demonstrate that nested scheduling is a promising concept for GPU scheduling in nested environments. We discuss possible future improvements in the next chapter.

6 Conclusion

In this thesis, we presented two approaches on GPU scheduling in virtualized environments. Our primary approach supplies a central GPU scheduler in the hypervisor with paravirtual hints about guest tasks in the virtual machines. The scheduler uses these hints to manipulate the virtual machine's memory mappings of GPU channels belonging to individual guest tasks. This technique allows for independent accounting and scheduling of VM guest tasks. To the best of our knowledge, this is the first approach that is able to simultaneously schedule GPU usage in multiple levels of virtualization.

Our second approach is fully decentralized and employs multiple schedulers: One in the host and one in each virtual machine. It is each scheduler's responsibility to enforce fairness locally between child tasks of their machine.

We used two existing approaches for GPU virtualization and GPU scheduling as a basis in order to implement a full prototype of our paravirtual approach, which is able to enforce fair GPU sharing both between and inside virtual machines at the same time. We also implemented a proof of concept of our nested scheduling approach.

In the evaluation, we measured an average scheduling overhead of only 2.17 % with our paravirtual approach. Afterwards, we tested fairness in two different scenarios. The first scenario runs benchmarks together with an increasingly greedy throttle application. Here, our scheduler is able to enforce fair GPU sharing with all benchmarks, which demonstrates the ability to schedule between guest tasks running inside virtual machines. In the second scenario, we ran the benchmarks on the host and an increasing number of throttle instances in one virtual machine. This scenario demonstrates the ability to treat entire virtual machines as one scheduling entity and enforce fairness between host tasks. Additionally, we demonstrated the basic feasibility of decentralized GPU scheduling with our nested prototype. We showed that it is possible to account and schedule the GPU usage of applications purely from within a virtual machine without any help from the hypervisor.

6.1 Future Work

One natural next step is a full design and implementation of our nested approach. The most important design aspect of such a decentralized scheduling system is the organization of sampling. We gave a brief introduction into this complex topic in Section 3.4.1, but more work is required in order to solve this problem properly.

Another direct followup is the extension of our prototype to support multiple layers of virtualization, that is, virtual machines inside of virtual machines. For the sake of simplicity, we limited ourselves to one layer in the implementation. In the design chapter, however, we already covered many aspects of nested virtualization. Consequently, we see this primarily as an implementation effort.

We also see potential for a hybrid approach, which combines ideas from our paravirtual and nested approaches. A clever combination of both approaches largely avoids sampling interactions, the core difficulty of the nested approach, but retains its flexibility of allowing children to run their own scheduling. One possible idea is to keep the paravirtual components and do all accounting in the hypervisor, but only enforce inter-VM fairness there. Then pass the accounting results to the virtual machines, where a nested scheduler uses the information to enforce local fairness according to each VM's individual policy.

Another important design aspect is the handling of malicious applications, that not only overuse their fair share, but try to reach denial-of-service by sending kernels with unlimited runtime. Since running kernels cannot be preempted, GPU schedulers in general have difficulties dealing with such clients. The original version of NEON allows to configure a maximum kernel runtime, after which the issuing application is killed. This works, because the GPU driver is able to abort kernel executions on task exits. In an virtualized environment, however, the situation is more complex, since there is no easy way for the hypervisor to kill malicious guest tasks. As a last resort, the hypervisor can kill the whole virtual machine that contains the malicious task, but this would be a rather excessive response. A less severe solution is to design a mechanism which allows the hypervisor to inform the virtual machine about the malicious guest application, in order to give the VM a chance to deal with the problem itself. However, if the virtual machine does not cooperate in removing the malicious application, the hypervisor still needs to stop the entire VM.

As a final idea, one could investigate how to include support for scheduling priorities. As the number of applications increases, fine-grained scheduling control via priorities becomes more and more interesting. As an example, a background application that uses the GPU and runs on the host of a larger virtualization hierarchy receives the same amount of GPU computation time as a full virtual machine. Scheduling priorities allow a more fine-grained configuration of the target share of such processes. With the scheduling weights used in our paravirtual approach, we already have a promising mechanism that can probably be extended for full support of scheduling priorities.

Bibliography

[1] K. Adams and O. Agesen. “A Comparison of Software and Hardware Techniques for x86 Virtualization”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA, 2006, pp. 2–13. DOI: 10.1145/1168857.1168860.

[2] Advanced Micro Devices. *Multiuser GPU*. URL: http://www.amd.com/Documents/Multiuser-GPU-Datasheet.pdf (visited on 07/29/2016).

[3] Advanced Micro Devices. *Virtualization*. URL: https://www.amd.com/en-us/solutions/professional/virtualization (visited on 07/29/2016).

[4] M. Bautin, A. Dwarakinath, and T.-c. Chiueh. “Graphic engine resource management”. In: *Proc. SPIE* 6818 (Jan. 2008): *Multimedia Computing and Networking 2008*. DOI: 10.1117/12.775144.

[5] G. Bosilca, H. Ltaief, and J. Dongarra. “Power profiling of Cholesky and QR factorizations on distributed memory systems”. In: *Computer Science - Research and Development* 29.2 (2014), pp. 139–147. DOI: 10.1007/s00450-012-0224-2.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IISWC 2009. Austin, Texas, USA, Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[7] H. Chen, L. Shi, and J. Sun. “VMRPC: A high efficiency and light weight RPC system for virtual machines”. In: *Proceedings of the 18th IEEE International Workshop on Quality of Service*. IWQoS 2010. Beijing, China, June 2010, pp. 1–9. DOI: 10.1109/IWQoS.2010.5542746.

[8] D. Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008. ISBN: 978-0133582499.

[9] A. Demers, S. Keshav, and S. Shenker. "Analysis and Simulation of a Fair Queueing Algorithm". In: *Symposium Proceedings on Communications Architectures & Protocols*. SIGCOMM '89. Austin, Texas, USA, 1989, pp. 1–12. DOI: 10.1145/75246.75248.

[10] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters". In: *Proceedings of the 8th International Conference on High Performance Computing and Simulation*. HPCS 2010. Caen, France, June 2010, pp. 224–231. DOI: 10.1109/HPCS.2010.5547126.

[11] *Envytools*. URL: https://envytools.readthedocs.io/en/latest/# (visited on 07/30/2016).

[12] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling". In: *38th Annual International Symposium on Computer Architecture*. ISCA'11. San Jose, CA, USA, June 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[13] J. Fang, A. L. Varbanescu, and H. Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. ICPP-2011. Taipei City, Sept. 2011, pp. 216–225. ISBN: 978-0123838728. DOI: 10.1109/ICPP.2011.45.

[14] M. Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004. ISBN: 978-0-321-19368-1.

[15] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds". In: *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*. Euro-Par 2010. Ischia, Italy: Springer Berlin Heidelberg, 2010, pp. 379–391. DOI: 10.1007/978-3-642-15277-1_37.

[16] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. "LoGV: Low-Overhead GPGPU Virtualization". In: *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*. HPCC-EUC 2013. Zhangjiajie, Hunan Province, China, Nov. 2013, pp. 1721–1726. DOI: 10.1109/HPCC.and.EUC.2013.245.

[17] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. “GViM: GPU-accelerated Virtual Machines”. In: *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. HPCVirt ’09. Nuremburg, Germany, 2009, pp. 17–24. DOI: 10.1145/1519138.1519141.

[18] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. “Pegasus: coordinated scheduling for virtualized accelerator-based systems”. In: *Proceedings of the 2011 USENIX Annual Technical Conference*. USENIX ATC’11. Portland, Oregon, USA, 2011, p. 31. ISBN: 978-931971-85-0. URL: https://www.usenix.org/legacy/events/atc11/tech/final_files/atc11_proceedings.pdf#page=41 (visited on 09/07/2016).

[19] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. “42 TFlops Hierarchical N-body Simulations on GPUs with Applications in Both Astrophysics and Turbulence”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon, USA, Nov. 2009, 62:1–62:12. DOI: 10.1145/1654059.1654123.

[20] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[21] Intel Corporation. *Desktop 4th Gen Intel Core Processors Spec Update*. URL: https://www-ssl.intel.com/content/www/us/en/processors/core/4th-gen-core-family-desktop-specification-update.html (visited on 09/05/2016).

[22] Intel Corporation. *GVT-g Kernel*. URL: https://github.com/01org/igvtg-kernel (visited on 07/27/2016).

[23] Intel Corporation. *GVT-g QEMU*. URL: https://github.com/01org/igvtg-qemu (visited on 07/27/2016).

[24] S. Kato. *gdev-bench*. URL: https://github.com/shinpei0208/gdev-bench (visited on 09/09/2016).

[25] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. “TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments”. In: *Proceedings of the 2011 USENIX Annual Technical Conference*. USENIX ATC’11. Portland, Oregon, USA, 2011, p. 17. ISBN: 978-931971-85-0. URL: https://www.usenix.org/legacy/events/atc11/tech/final_files/atc11_proceedings.pdf#page=27 (visited on 09/07/2016).

[26] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. “Gdev: First-class GPU Resource Management in the Operating System”. In: *Proceedings of the 2012 USENIX Annual Technical Conference*. USENIX ATC’12. Boston, MA, USA, June 2012, pp. 401–412. ISBN: 978-931971-93-5. URL: `https://www.usenix.org/conference/atc12/technical-sessions/presentation/kato` (visited on 09/07/2016).

[27] K. Menychtas. *NEON-Linux kernel interface README.md*. URL: `https://github.com/KonstantinosMenychtas/neon/blob/master/linux-3.4.7/README.md` (visited on 09/10/2016).

[28] K. Menychtas, K. Shen, and M. L. Scott. “Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. Salt Lake City, Utah, USA, Mar. 2014, pp. 301–316. DOI: `10.1145/2541940.2541963`.

[29] K. Menychtas, K. Shen, and M. L. Scott. “Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack”. In: *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX ATC’13. San Jose, CA, USA, June 2013, pp. 291–296. ISBN: 978-1-931971-01-0. URL: `https://www.usenix.org/node/174532` (visited on 09/07/2016).

[30] K. Nance, M. Bishop, and B. Hay. “Virtual Machine Introspection: Observation or Interference?” In: *IEEE Security & Privacy* 6 (5 Sept. 2008), pp. 32–37. DOI: `10.1109/MSP.2008.134`.

[31] Nouveau Community. *Nouveau: Accelerated Open Source driver for nVidia cards*. URL: `https://nouveau.freedesktop.org/wiki/` (visited on 08/07/2016).

[32] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. “A detailed GPU cache model based on reuse distance theory”. In: *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*. HPCA-20. Orlando, Florida, USA, Feb. 2014, pp. 37–48. DOI: `10.1109/HPCA.2014.6835955`.

[33] NVIDIA Corporation. *GeForce Titan Black Specification*. URL: `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-black/specifications` (visited on 07/27/2016).

[34] NVIDIA Corporation. *Multi-Process Service*. URL: `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf` (visited on 08/21/2016).

[35] NVIDIA Corporation. *NVIDIA GRID vGPU and VMware Horizon*. URL: `https://images.nvidia.com/content/grid/vmware/nvidia-grid-vmware.pdf` (visited on 07/29/2016).

[36] NVIDIA Corporation. *Virtual GPU Technology for Hardware Acceleration*. URL: `https://www.nvidia.com/object/grid-technology.html` (visited on 07/29/2016).

[37] Nvidia Corporation. *CUDA C Programming guide*. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visited on 07/29/2016).

[38] D. Ongaro, A. L. Cox, and S. Rixner. "Scheduling I/O in Virtual Machine Monitors". In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '08. Seattle, WA, USA, 2008, pp. 1–10. DOI: `10.1145/1346256.1346258`.

[39] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899. DOI: `10.1109/JPROC.2008.917757`.

[40] PathScale Inc. *pscnv - PathScale NVIDIA graphics driver*. URL: `https://github.com/pathscale/pscnv` (visited on 08/07/2016).

[41] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. "PTask: Operating System Abstractions to Manage GPUs As Compute Devices". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal, 2011, pp. 233–248. DOI: `10.1145/2043556.2043579`.

[42] L. Shi, H. Chen, J. Sun, and K. Li. "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines". In: *IEEE Transactions on Computers* 61.6 (June 2012), pp. 804–816. DOI: `10.1109/TC.2011.112`.

[43] T. Shimokawabe et al. "An 80-fold speedup, 15.0 TFlops full GPU acceleration of non-hydrostatic weather model ASUCA production code". In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. New Orleans, LA, USA, Nov. 2010, pp. 1–11. DOI: `10.1109/SC.2010.9`.

[44] W. R. Stevens and S. A. Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2005. ISBN: 9780201433074.

[45] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in science & engineering* 12 (3 2010), pp. 66–72. DOI: 10.1109/MCSE.2010.69.

[46] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. “GPUvm: Why Not Virtualizing GPUs at the Hypervisor?” In: *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX ATC’14. June 2014, pp. 109–120. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki (visited on 09/07/2016).

[47] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. “Towards Multi-tenant GPGPU: Event-driven Programming Model for System-wide Scheduling on Shared GPUs”. In: *2016 Workshop on Multicore and Rack-scale Systems*. MaRS’16. London, UK, Apr. 2016. URL: https://www.cs.utexas.edu/~mars2016/workshop-program/MaRS_2016_paper_6.pdf (visited on 09/07/2016).

[48] K. Tian, Y. Dong, and D. Cowperthwaite. “A Full GPU Virtualization Solution with Mediated Pass-Through”. In: *Proceedings of the 2014 USENIX Annual Technical Conference*. Philadelphia, PA, USA, June 2014, pp. 121–132. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/tian (visited on 09/07/2016).

[49] I. S. Ufimtsev and T. J. Martínez. “Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation”. In: *Journal of Chemical Theory and Computation* 4.2 (2008), pp. 222–231. DOI: 10.1021/ct700268q.

[50] *XML-RPC*. URL: http://www.xmlrpc.com/ (visited on 07/27/2016).