

Tell me when you are sleepy and what may wake you up!

Djob Mvondo, Antonio Barbalace, Alain Tchana, Gilles Muller

▶ **To cite this version:**

Djob Mvondo, Antonio Barbalace, Alain Tchana, Gilles Muller. Tell me when you are sleepy and what may wake you up!. SoCC 2021 - ACM Symposium on Cloud Computing, Nov 2021, Seattle WA USA, United States. pp.562-569, 10.1145/3472883.3487013 . hal-03503825

HAL Id: hal-03503825
https://hal.science/hal-03503825

Submitted on 3 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Tell me when you are sleepy and what may wake you up!

Djob Mvondo, Antonio Barbalace
The University of Edinburgh
{djob.mvondo,antonio.barbalace}@ed.ac.uk

Alain Tchana
ENS Lyon
alain.tchana@ens-lyon.fr

Gilles Muller
INRIA
gilles.muller@inria.fr

Abstract

Nowadays, there is a shift in the deployment model of Cloud and Edge applications. Applications are now deployed as a set of several small units communicating with each other – the microservice model. Moreover, each unit – a microservice, may be implemented as a virtual machine, container, function, etc., spanning the different Cloud and Edge service models including IaaS, PaaS, FaaS. A microservice is instantiated upon the reception of a request (e.g., an http packet or a trigger), and a rack-level or data-center-level scheduler decides the placement for such unit of execution considering for example data locality and load balancing. With such a configuration, it is common to encounter scenarios where different units, as well as multiple instances of the same unit, may be running on a single server at the same time.

When multiple microservices are running on the same server not necessarily all of them are doing actual processing, some may be busy-waiting – i.e., waiting for events (or requests) sent by other units. However, these "idle" units are consuming CPU time which could be used by other running units or cloud utility functions on the server (e.g., monitoring daemons). In a controlled experiment, we observe that units can spend up to 20% - 55% of their CPU time waiting, thus a great amount of CPU time is wasted; these values significantly grow when overcommitting CPU resources (i.e., units CPU reservations exceed server CPU capacity), where we observe up to 69% - 75%. This is a result of the lack of information/context about what is running in each unit from the server CPU scheduler perspective.

In this paper, we first provide evidence of the problem and discuss several research questions. Then, we propose an handful of solutions worth exploring that consists in revisiting hypervisor and host OS scheduler designs to reduce the CPU time wasted on idle units. Our proposal leverages the concepts of informed scheduling, and monitoring for internal and external events. Based on the aforementioned solutions, we propose our initial implementation on Linux/KVM.

1 Introduction

Recently, more and more Cloud and Edge customers are switching from deploying their applications as a monolith to the microservice model, in which applications are deployed as an ensemble of different small units – the microservices, which communicate with each other over the network. Virtualization is used as a core technology to power the microservices, enabling many of them to run on the same machine in isolation, thus securely, even when they belong to different Cloud or Edge customers.

Each microservice can be implemented as a virtual machine, container, function, etc., spanning IaaS, PaaS, and FaaS deployments. IaaS uses "heavyweight" full-fledged virtual machines (VM) while FaaS relies on "lightweight" VMs, such as microVMs [5]. A FaaS VM runs a very limited number of tasks (potentially, only one function) compared to a IaaS VM that may run several applications.

A critical component of a virtualization platform is the **scheduler**. In both IaaS and FaaS, the hypervisor or host OS -level scheduler plays a key role to achieve high throughput and low I/O latency and meet service level objectives (SLOs). However, several research works pinpoint that current hypervisor or host OS -level schedulers do not perform as expected when faced with specific workloads. For instance, facing mix workloads composed of CPU and I/O intensive tasks, the scheduler exhibits priority inversion issues[18, 34] and unnecessary I/O stalls[19]. Another problematic workload consists of spinlock intensive applications which experience the lock holder and the lock waiter[21, 30, 36, 37, 40] preemption. When analysing these issues, we observe that their root cause is the *scheduler hierarchy combined with the black box nature of VMs*, which introduces a semantic gap. The guest level scheduler thinks that it directly controls the hardware, whereas the hardware is controlled by the hypervisor or host OS, which in turn is not aware of the decisions taken by the guest software.

In this paper, we demonstrate for the first time that the same kind of problems severely reduce the number of microservices that can be co-placed on the same server, or drastically increase their serving latencies. Due to insufficient knowledge about the microservices running in VMs and the obliviousness of the way microservices communicates, current hypervisor or host OS -level schedulers lead to the issue that we call **avoidable** latencies between microservices' triggers, or events, and their actual execution. For example, in a FaaS scenario, when Firecracker[1] faces the execution of a chain of functions on the same physical server, the scheduler do not consider functions' nature to affect execution order, thus resulting in unpredictable trigger and execution times (see §2). We noticed that servers tend to spend a great amount of time fairly scheduling the different functions, while they are barely waiting for inputs – up to 75% of functions' total CPU time in a controlled experiment.

Background & Related Work. When no virtualization is involved, an operating system (OS) scheduler running on

bare-metal decides what task executes next on the available processing units. Traditional OSes adopt a fair scheduling algorithm that guarantees an "equal" time of execution per task[9]. On a server that uses virtualization, two or more schedulers are stacked: the bottom-level hypervisor's or host OS' scheduler (e.g., credit in Xen), and the upper-level guest OS's scheduler (e.g., CFS in Linux). The bottom-level scheduler acts similarly to OS schedulers running on bare-metal except that, instead of tasks, it deals with virtual CPUs (vCPUs). In most of the cases, the guest OS is a black-box – hence, the bottom-level and upper-level scheduler do not communicate.

Several research works attempt to address issues related to hierarchical scheduling both in bare-metal and virtualized systems.

In bare-metal OSes, existing works examine scheduling issues, focusing on thread blocking. Scheduler activations [4], which introduced N:M scheduling decades ago, provide a kernel-user mechanism to make the user-level scheduler switch to another user-level thread when a user-level thread blocks in the kernel. Although there is no virtual machine involved, it solves a hierarchical scheduling problem. Linux Futex[11, 16], is more recent kernel-user mechanism – without VMs, that inform the kernel about the spinning state of an application – thus, avoiding wasting CPU resources.

Regarding IaaS, some approaches either exploit vCPUs migration between pCPUs to reduce I/O workloads stalls [19], ballooning to reduce priority inversion issues [34], or combining CPU hardware features to reduce the cost of vCPUs context-switches [18]. Teabe et al. [35] modified both the hypervisor and the guest OSes schedulers such that, a guest OS scheduler schedules a task that wants to take a lock only if the remaining quantum is enough to perform the critical section, thus avoiding both the lock holder and the lock waiter preemption problems. Regarding FaaS, research works tend to focus on inter-server scheduling policies for dispatching functions among worker nodes to improve load balancing and data locality [20, 31, 42]. To the best of our knowledge, no work targets intra-server scheduling issues in FaaS.

Overall, prior works require significant modifications of the guest OS level scheduler and are rigid, i.e., they are tightly related to a specific issue. Thus, cannot correctly handle diverse scenarios. *We argue that a more generic approach is needed to handle constantly changing workloads as encountered in IaaS and FaaS environments today* – e.g., microservices scenario. Besides, we think that it is urgent to address this problem as nested virtualization is gaining in popularity[26].

Finally, other solutions to the hierarchical scheduling problem span from flattened scheduling to introspection and guest live-modifications [7, 10, 12, 33] or include runtime scheduler extensions, such as proposed by Small et al. [33], by extOS [7], or Ipanema [25]. This work builds on those.

Contribution. In this paper, we propose a redesign of the traditional hypervisor, or host OS, scheduling in data centers, which include (a) accompanying VMs/tasks with contextual knowledge about their working model, for example in terms of network activity; (b) monitoring VMs/tasks for events that change their internal and external state. The redesign is motivated by a campaign of experiments targeting microservices implemented in FaaS (§2), whose experimental results are explained in §3. We discuss several possible approaches while unveiling our design (§4) and conclude in §5.

2 Motivations

To fully understand the problem at hand, we set up a controlled experiment that aims at highlighting the core issues with scheduling. To this aim, we chose to trigger the execution of several inter-dependent workloads on a server. Inter-dependent workloads are extensively deployed in data centers, including microservices, tenants I/O-bound applications, parallel multiprocessing (e.g., MapReduce or MPIs) jobs, machine learning training jobs, etc.

We arrange our microservice workloads to be implemented as FaaS. Our choice is motivated by the growing interest in both industry and academia on FaaS. FaaS-based applications consist of several functions that are called in sequence or graph. FaaS platforms mainly involve scheduling event handlers and the running functions. Most FaaS platforms (e.g., OpenWhisk[29], Firecracker, Knative[24]) support the concept of *chain or pipeline of functions* ($F_{i,i=1,\ldots,n}$), where after the trigger of the first function in the chain (F_1), the remaining functions are sequentially and automatically triggered. Usually, the payload of function $F_{i,i\neq 1}$ in a chain, includes the output of $F_{i-1,i\neq 1}$. These chains are often used to express workflows use cases[1] and represent up to 31% of serverless functions in the Cloud[14].

However, when faced with several chains of functions on a single server, especially on an overcommitted server, scheduling becomes a cumbersome task (at the hypervisor or host OS -level). *How chains should be prioritized among each other to achieve agreed service-level latencies?* The complexity is exacerbated when (micro-)VMs[2] are used to host functions (which is the case with AWS Lambda) due to the blackbox nature of VMs. To assess this problem, we design the following experiment(s).

2.1 Experimental Scenario

Despite microservices naturally run on several different machines – i.e., distributed, it is the case that the microservices part of a single workflow may be placed by a data-center scheduler on a single machine. Hence, and also in order to better quantify the problem at end, we herein focus on chains of microservices on a single server. To ease deployment, we decided to use Amazon Firecracker. However, we believe that the same results apply to other VM technologies, such as Kata Containers [22]. Lately, in Section 2.2, we reassess our results within a FaaS container-based scenario.

[1] Checkout some examples at: https://doi.org/10.5281/zenodo.3862625
[2] When compared to containers

We chose chains of 5 and 3 microservices based on [14], which states that 82% *of all use cases consist of applications that use five or less different functions.* Specifically, we compute 2 chains applications. The first one performs image processing and consists of 5 image processing functions (blurring, edging, resize, gamma, and sepia filtering) to generate a thumbnail, and the second one performs online compiling with gg[15] and does a 3-stage compilation of a hello world in C (denoted make) and llvm build. Our applications come from the ServerlessBench suite[41].

We are interested in the following function-level metrics: ① the latency between the trigger and the start of the chain execution (*trigTime*), ② the inter function latency[3] (*avgInterTime*), and ③ the total chain execution time (*execTime*). For the image processing application, the input[4] and output images are read and stored from-to AWS S3.

We then repeat the same experiment with the image processing application whilst increasing the number of colocated chains, which are all triggered at the same time. The number of colocated chains goes from 0 to 50. Additionally, for every run, we collect the following low-level scheduling metric. ④ the idle time inside each micro-VM created during the experiment (*idleTime*) i.e., the overall time a micro-VM is scheduled on a CPU but the function within it cannot do any processing either because it awaits an I/O, or it awaits input from another function as illustrated in Figure 1.

Experimental Setup. For a fair evaluation, we run the colocation related experiments under two network tail latency scenarios; the first — *local network* — the functions run in a `a1.metal` server on AWS EC2 whilist the second — *remote network* — the functions run in our `in-lab` server (see below), thus the network latency to AWS S3 are higher compared to the first situation.

The server that we used in our lab is a PowerEdge R430 with an Intel Xeon E5-2620 v4 (8 cores/16 hyperthreads at 2.10GHz), 16 GB memory, 1TB hard drive, and 2 NetXtreme BCM5720 1Gbps. On AWS EC2, an a1.metal server instance has 16 physical cores (Custom built AWS Graviton Processor with 64-bit Arm Neoverse n1), 32 GB memory, and up to 10Gbps network bandwidth. The two servers run Linux 4.19.0-13-amd64 and microVMs are powered with Firecracker v0.24.0. Thus, our experiments cover a CPU overcommit ratio (vCPU:pCPU) of 1:1, 3:1, 6:1, as supported in production environments[8, 23, 27, 32, 38].

2.2 Experimental Results

Figure 2 and 3 presents our evaluation results. On a single chain invocation (2), we observe that the units CPU idle times ratio ranges between 16% to 27% of the total CPU time used by the application. These numbers worsen whenever we increase the over commitment ratio.

[3]The average time it takes to schedule the next function in the chain
[4]The input images are 256KB and 1MB in size

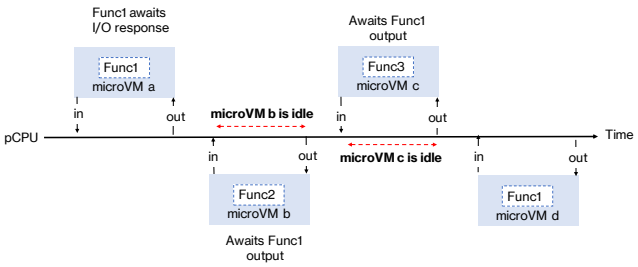


Figure 1. *Illustration of micro-VMs idle times. Micro-VMs b and c running Func2 and Func3 respectively, are scheduled even though they await Func1 output which has not finished running. This results in wasted CPU time.*

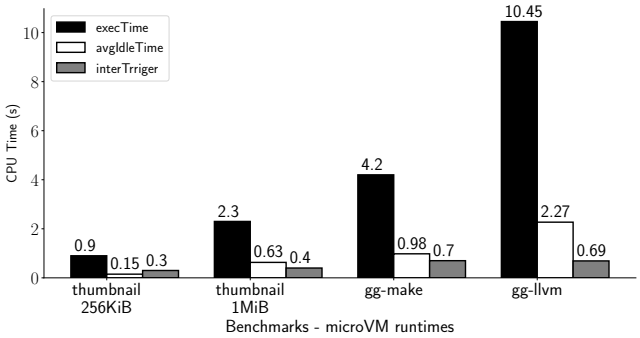
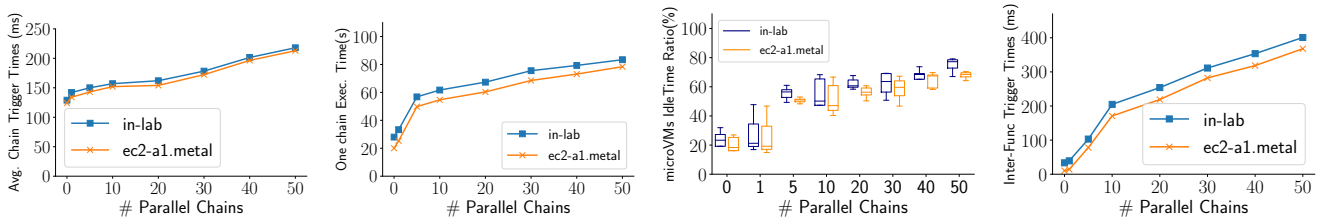


Figure 2. *Total CPU times, idle CPU times, and average inter trigger times for 2 pipelines: image processing and online compiling. The runs are performed on AWS EC2* `a1.metal`.

As shown in Figure 3, the total execution time of the image processing chains hugely increases with the number of parallel invocations, ranging from 28.3*s* up to 83.41*s* with the `in-lab` setup and 20*s* up to 78.52*s* with the `a1.metal` setup, thus an overhead of 2.97*x* and 3.92*x* for `in-lab` and `a1.metal` respectively. *The observed overhead is a result of bad scheduling decisions* which lead micro-VMs to being *scheduled while the function within it cannot do any processing*. Indeed, the average micro-VMs idle times' ratio[5] as shown in Figure 3-c ranges from 20.18% to 75.31% with the `in-lab` setup and 18.25% to 69.25% with the `a1.metal` setup, thus an increment factor of 3.73*x* and 3.79*x* for the `in-lab` and `a1.metal` setup respectively. These idle times undermine chain trigger times (Figure 3-a) and intra-chain function trigger times (Figure 3-d), which overall affects the total chains' execution times.

What about containers? As someone would expect, the observed wasted CPU cycles are curtailed when running microservices within containers (OS-level virtualization). Since a container is a set of OS' processes, the OS scheduler has more insight on whether a process is for example spinning or awaiting some external event, therefore, it is in the position to reduce the idle times' ratio. We confirmed that by rerunning the same experiments introduced above using Apache OpenWhisk[29] in standalone mode (v1.0.0). Apache OpenWhisk is an open source FaaS platform that leverages Docker

[5]With respect to each micro-VM total runtime



(a) *Average chain trigger times (trigTime) per parallel functions invocations.*

(b) *One chain total execution time (execTime) per parallel functions invocations.*

(c) *microVMs idle times ratio (idleTime) distribution per parallel functions invocations.*

(d) *Average inter-func trigger times (avgInterTime) per parallel functions invocations.*

Figure 3. *Colocated chains experiment: for every run, we plot the results obtained with our* `in-lab` *server and AWS EC2* `a1.metal` *instance.*

containers to run functions. We observed container idle time ratios ranging from 17.8% to 31.6% (over commitment experiment), thus a lower CPU time waste ratio compared to micro-VMs.

Despite containers are known to be less secure than (micro-) VMs – in fact, FaaS providers (such as Azure Functions) which run functions inside containers, stack them into VMs for security purposes [39], and Amazon runs a function per micro-VM, both containers and (micro-) VMs are part of Cloud and Edge providers offering today. Our experiments show that both technologies are prone to the same scheduling-related issue, while the effect is more demarcate on the "more secure" technology.

3 Unfolding the Puzzle

As already mentioned, what we identified with these experiments is not fully new in the virtualization realm, but another instance of a known problem: the existence of a semantic gap between the scheduler in the hypervisor/host OS, and the scheduler in the VM (or what the application does, in the case of containers). The same problem manifested before in the OS literature, as early as when user-level threading has been integrated with kernel-level threading – e.g., schedule activations [4]. Herein, we are addressing the same high level problem(s), but with a broader scope – i.e., not restricted to inactive VMs that just busy wait. At the same time we propose to use existing contextual information, in this case, how several microservices are chained together and the data that they exchange.

Our work is based on the followings observations.

- First, a microservice may not be doing any useful processing (idle or inactive) not just when it is busy waiting.
- Second, we noticed that with microservices, which communicate mostly with network packets, it is straightforward to identify what events may turn an idle or inactive microservice into an active one.

We further dig into these two issues below.

I may sleep! Microservices are commonly scheduled with an interactive and fair scheduling algorithm on a single machine, such as CFS in Linux, or credit scheduler in Xen. Microservices can be implemented atop an operating system, e.g., in containers, or as guest VM deployed as a unikernel, or within a traditional operating system with full or stripped operating environment, which is what we used in our evaluation (microVM).

Independently of the deployment, a vCPU that is busy waiting on a spinlock can be de-scheduled until the target memory area is updated – this has been covered in previous literature [18, 19], and it doesn't affect only VMs, with a full and stripped down operating system, but also unikernels, and containers whose applications exploits kernel-bypass technologies (e.g., DPDK/SPDK). In all such cases, for performance reasons, an application may busy waiting for events, without notifying any other level software.

However, especially when a traditional operating system runs a microservice, such microservice may not be processing any request and still no software is busy waiting. Instead, the software on the VM may carry on management tasks at the user or OS-kernel level, which may or may not include an idle loop. Hence, detecting software that is busy waiting is not the only way to conclude that a VM may be put to sleep waiting for an event. In fact, a VM may be also put to sleep right after the scheduler start scheduling the idle task, or when a microservice sends a response to a previous request. Obviously, the microservice software in the VM may also explicitly inform the scheduler – but this is not implemented yet, and it require trusting the microservice, which may be an attack vector.

Wake me up! Other than detecting when a VM doesn't need to be scheduled because not doing any useful work, it is fundamental to identify when it should be put back to run. External events that may put back a VM into run are: modification of a memory area, modification of a device mapped area due for example to end of a data transfer (network packet, storage blocks, etc), device interrupt, timer interrupt, IPI (for SMP), management interrupt (ipmi, smbus, acpi, error, etc.). Some of these events have been demonstrated before that they can be postponed [3, 6], including timer interrupts in most of the

cases, while uniprocessor VMs have significantly less events – management interrupts are at a minimum for Cloud VMs.

Although such events may be delayed or coalesced, they must be monitored, and the scheduler must know which particular set of events should put back a VM to run – such as a network request to a microservice. Thus, for the hypervisor or host OS scheduler, it must be possible to monitor such events and move a VM back to "normal" scheduling (e.g., to a CPU ready/run queue) when specific events happen. Finally, each VM is concern in specific events, which must be detailed at some level.

When focusing on network communication, which is the main way microservices use to communicate, the hypervisor or host OS scheduler should look up for packets flowing to and from the VM. Although this is possible with paravirtualized devices, or traditional network stack for containers, this become not obvious when SR-IOV or DPDK/SPDK are adopted, because such technologies are supposed to bypass the hypervisor or host OS. Finally, it is important to define a way to inform the scheduler about what events should wake up a microservice, and despite this can be done by the user itself, this maybe a security risk if not done in a controlled way.

Key Takeaway. To keep packing more and more microservices on the same machine, without largely affecting their service latency as shown by our experiments, we need to *rethink how scheduling works*. While what have been proposed previously in the literature may cover some cases, it is not enough to support the microservice scenarios presented here. Contextual information, which changes over time, must be obtained or provided to the hypervisor/host OS scheduler, which should be notified by what events are happening in the system. Finally, security shouldn't be overlooked.

4 Approaches

We propose that scheduling decisions in a hypervisor, or host OS, should be taken by considering contextual information regarding the running software, especially when a semantic gap exists. In the microservice scenario, the contextual information consists in how different microservices are chained and what messages they exchange. At high level, we envision hypervisors' or OS' schedulers to be customizable per-schedulable-entity (or eventually, user). Customization is fundamental to reduce the priority of a schedulable entity[25, 28] – when it is not going to do any actual work, and increase the priority of the latter because it has suddenly received work. Scheduler customizations are not enough per-schedulable entity, additional mechanisms to identify idle or wake up events related to a VM/task are needed. Because identification mechanisms have been widely researched before, including VM and packet introspection [2, 10, 12], as well as machine learning [6], herein we focus on how the scheduler should be customized and how to communicate events to it. Lastly, due to our initial use-case scenario, chains of microservices that run in multi-tenancy environments, security is at the highest stake, and it has been given high consideration in our design.

To achieve our goal, *is it really necessary to modify the hypervisor's or OS's scheduler?* In fact, it is not strictly necessary to do that. In a KVM-like environment, a user-space control program may collect the contextual information, monitor the execution of the VMs for events, and change the scheduling priority of VMs accordingly. However, such solution likely has a very high overhead originating by the large amount of context switches between the control program, the kernel and the VM/task being monitored. Nevertheless, in a type-1 hypervisor scenario the control program maybe running in a DOM0-like VM, but there is no easy way to have such control program not affecting the hypervisor scheduler. Hence, we believe that the aforementioned mechanisms should be put into the hypervisor or host OS kernel itself – and not only for a performance reason. We foresee the following three approaches to informing the scheduler of contextual information.

Approach 1: Directed by a trusted source. A trusted source, i.e., the Cloud orchestrator, provides to a server together with the target VMs, the information that can be fed to the hypervisor or host OS scheduler. Thus, the information fed to the scheduler is not coming directly from the user and the host-OS or hypervisor can trust it.

The scheduling information may be provided in a manifest file, as a set of rules in a domains specific language, or as a precompiled program/plugin written in a restricted ISA (e.g., eBPF, WebAssembly). However, the former requires the development and deployment of a program to read and interpret the manifest file(s), while the latter an interpreter or JIT compiler for the safe execution of the code.

Approach 2: Fully Collaborative. Similar to the previous approach, but it is the VM that at runtime provides the contextual information to the hypervisor's or host OS' scheduler of the server. A major limitation of this approach is security. Security is at a stake here because the information provided by the VM itself is not fully trustable. The guests may use this as a vector to construct a DoS attack to the server.

Approach 3: Learning. The scheduler on the host OS, or hypervisor, instead of having the contextual information being provided, tries to reconstruct it itself, with a form of VM introspection. Specifically, it monitors the guest VM in order to identify if it is actually producing any observable external state (e.g., requesting or transmitting data to/from the user). The host OS or hypervisor uses such information to change the interactivity with the target VMs. This is similar to what interactive scheduling algorithms (the ones that consider IO when taking decisions) do, but applied to the problem of scheduling processes in a hierarchy scheduling environment. The main drawback of this method lies in the fact that if you don't know the guest or cannot inspect it, this is not a viable option, e.g., encrypted VMs, such as AMD SEV.

4.1 Prototype Design

With the goal of oversubscribing CPU resources (3:1 and up) in microservice deployments, but without affecting the service latencies, we are developing an extensible hypervisor/host OS scheduler that can be customized at runtime, and the customization information are shipped together with the VM and automatically built by the Cloud provider. For the time being, we assume that the Cloud provider has a mechanism that either enforce the user to specify network request/response pairs or automatically recognize those (e.g., via learning). Because such information is provided by the Cloud provider, it is trustable and used on a server to identify sleep/wake up events and therefore extend the scheduler.

At the moment, we are targeting Linux/KVM and Firecracker microVMs. Hence, we decided to extend the Linux kernel scheduler (host OS) with eBPFs. eBPFs in the scheduler communicate with eBPFs in the network layer and by monitoring network packets enable better scheduling decisions. eBPF are forged by the Cloud provider based on the mentioned knowledge of request/response packet pairs. As the user is not involved in the creation of the eBPFs, we believe there are no adversarial security issues. Finally, we choose eBPF as a quick, highly-customizable solution with an increasing community[13, 17], but a similar technology, e.g., Webassembly, may be used.

5 Conclusion

We presented yet another scheduling-related problem in the host OS/hypervisor layer that manifests in modern Cloud and Edge data centers where applications are increasingly deployed within the microservice model. The problem is another instance of the issue of semantic gap where several layers of software are running on a single machines and there is no communication. Which is common for several reasons, including legacy support and security.

We highlight that such semantic gap can cost up to 69% or 75% wastage of CPU time, and investigate its root causes. Additionally, we sketch different approaches to solve this problem, which rely on the idea of providing additional context regarding each microservice, while outlining drawbacks of any approach in terms of security. Lastly, we describe a current work that aims at extending the host OS/hypervisor scheduler – Linux/KVM, via eBPF to get a runtime customizable scheduler, which can effectively detect when to sleep and wake up microservices.

Acknowledgments

We thank our shepherd, Amit Levy, and the anonymous reviewers for their insightful feedback.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Yuvraj Agarwal, Stefan Savage, and Rajesh Gupta. 2010. SleepServer: A Software-Only Approach for Reducing the Energy Consumption of PCs within Enterprise Environments. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 22.

[3] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. 2011. VIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) *(USENIXATC'11)*. USENIX Association, USA, 4.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, California, USA) *(SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 95–109. https://doi.org/10.1145/121132.121151

[5] AWS. 2018. Introducing Firecracker, a New Virtualization Technology and Open Source Project for Running Multi-Tenant Container Workloads. http://tiny.cc/iwm7tz. Online; accessed Jan, 05 2021.

[6] M. Bacou, G. Todeschi, A. Tchana, D. Hagimont, B. Lepers, and W. Zwaenepoel. 2019. Drowsy-DC: Data Center Power Management System. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 825–834. https://doi.org/10.1109/IPDPS.2019.00091

[7] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. 2019. ExtOS: Data-Centric Extensible OS. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hangzhou, China) *(APSys '19)*. Association for Computing Machinery, New York, NY, USA, 31–39. https://doi.org/10.1145/3343737.3343742

[8] Salman A. Baset, Long Wang, and Chunqiang Tang. 2012. Towards an Understanding of Oversubscription in Cloud. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (San Jose, CA) *(Hot-ICE'12)*. USENIX Association, USA, 7.

[9] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. 2018. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 85–96. https://www.usenix.org/conference/atc18/presentation/bouron

[10] K. Burns, A. Barbalace, V. Legout, and B. Ravindran. 2014. KairosVM: Deterministic introspection for real-time virtual machine hierarchical scheduling. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 1–8. https://doi.org/10.1109/ETFA.2014.7005061

[11] Jonathan Corbet. 2020. Rethinking the futex API. https://lwn.net/Articles/823513/.

[12] Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. 2016. A Flattened Hierarchical Scheduler for Real-Time Virtualization. In *Proceedings of the 13th International Conference on Embedded Software* (Pittsburgh, Pennsylvania) *(EMSOFT '16)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/2968478.2968501

[13] eBPF. 2021. eBPF Foundation. https://ebpf.io/foundation/. Online; accessed Sep, 10, 2021.

[14] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2020. A Review of Serverless Use Cases and their Characteristics. arXiv:2008.11110 [cs.SE]

[15] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of

Transient Functional Containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 475–488.

[16] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. [n.d.]. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux.

[17] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 487–501. https://www.usenix.org/conference/nsdi21/presentation/ghigoff

[18] Weiwei Jia, Jianchen Shan, Tsz On Li, Xiaowei Shang, Heming Cui, and Xiaoning Ding. 2020. vSMT-IO: Improving I/O Performance and Efficiency on SMT Processors in Virtualized Clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 449–463. https://www.usenix.org/conference/atc20/presentation/jia

[19] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis C. M. Lau, Yuexuan Wang, and Yuangang Wang. 2018. Effectively Mitigating I/O Inactivity in vCPU Scheduling. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 267–280. https://www.usenix.org/conference/atc18/presentation/jia

[20] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 158–164. https://doi.org/10.1145/3357223.3362709

[21] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2016. Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds. *SIGOPS Oper. Syst. Rev.* 50, 1 (March 2016), 9–16. https://doi.org/10.1145/2903267.2903271

[22] Katacontainers. [n.d.]. Katacontainers: The speed of containers, the security of VMs. https://katacontainers.io/.

[23] Kenneth van Surksum. 2012. Best Practices for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environments.

[24] Knative. 2018. Knative. https://knative.dev/. Online; accessed Jan, 16 2021.

[25] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. 2020. Provable Multicore Schedulers with Ipanema: Application to Work Conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 16 pages. https://doi.org/10.1145/3342195.3387544

[26] Jin Tack Lim and Jason Nieh. 2020. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 557–574. https://doi.org/10.1145/3373376.3378467

[27] Martin Hosken. 2018. Architecting a VMware vSphere® Compute Platform for VMware Cloud Providers.

[28] G. Muller, J.L. Lawall, and H. Duchesne. 2005. A framework for simplifying the development of kernel schedulers: design and performance evaluation. In *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*. 56–65. https://doi.org/10.1109/HASE.2005.1

[29] OpenWhisk. 2016. Apache OpenWhisk - Open Source Serverless Cloud Platform. https://openwhisk.apache.org/. Online; accessed Jan, 16 2021.

[30] Jiannan Ouyang and John R. Lange. 2013. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. *SIGPLAN Not.* 48, 7 (March 2013), 191–200. https://doi.org/10.1145/2517326.2451549

[31] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. 2021. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems* 114 (2021), 259 – 271. https://doi.org/10.1016/j.future.2020.07.017

[32] Red Hat. 2021. Overcommitting resources. http://tiny.cc/xdobtz.

[33] Christopher Small and Margo Seltzer. 1996. A Comparison of OS Extension Technologies. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/usenix-1996-annual-technical-conference/comparison-os-extension-technologies

[34] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis C. M. Lau. 2017. Preserving I/O Prioritization in Virtualized OSes. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 269–281. https://doi.org/10.1145/3127479.3127484

[35] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. 2017. The Lock Holder and the Lock Waiter Pre-Emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock). In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 286–297. https://doi.org/10.1145/3064176.3064180

[36] Boris Teabe, Alain Tchana, and Daniel Hagimont. 2016. Application-Specific Quantum for Multi-Core Platform Scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/2901318.2901340

[37] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. 2004. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3* (San Jose, California) *(VM'04)*. USENIX Association, USA, 4.

[38] Uma Panda. 2017. How to decide VMWare vCPU to physical CPU ratio. https://www.cloudpanda.org/blog-single/?id=76.

[39] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–145.

[40] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin. 2016. Dynamic Acceleration of Parallel Applications in Cloud Platforms by Adaptive Time-Slice Control. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 343–352. https://doi.org/10.1109/IPDPS.2016.77

[41] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280

[42] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1225–1240. https://www.usenix.org/conference/osdi20/presentation/zhu