Towards User-Programmable Schedulers in the Operating System Kernel

Djob Mvondo, Antonio Barbalace, Jean-Pierre Lozi, Gilles Muller

Towards User-Programmable Schedulers in the Operating System Kernel

Djob Mvondo
Univ Rennes, Inria, CNRS, IRISA

Antonio Barbalace
The University of Edinburgh

Jean-Pierre Lozi
Oracle Labs

Gilles Muller
Inria

1 Introduction

Most of today's Operating Systems (OSes) run several applications at the same time. An OS component, the scheduler, decides what application's thread to execute next on each CPU core/thread. Traditional OSes, such as Linux, FreeBSD, Darwin, and Windows implement an in kernel-space scheduler that may offer one or more scheduling policies. For instance, Linux comes with a few schedulers always built-in, including the Completely Fair Scheduler (CFS) and the Real-Time (RT), each supporting one or more scheduling policies, e.g., RT supports First-In-First-Out (FIFO) or Round-Robin (RR). The assortment of scheduling policies provided by traditional OSes is meant to provide acceptable performance for most workloads on different architectures, from embedded devices to large multicore machines – achieving genericity.

What's wrong with fixed policies? Yet, recent works [4, 6, 13, 20, 21, 24] have shown that such policies are subject to performance bugs or exhibit suboptimal scheduling behaviours on some architectures and workloads. At the same time, bespoken classes of application's crave for custom policies due to a semantic gap between the application itself and the scheduler, for example in hierarchical scheduling §2.

Patching the kernel. Patching the OS kernel scheduler may solve suboptimal scheduling behaviours, as well as growing the number of available scheduling policies. However, this is outside reach for the average user, which is likely not familiar with programming OS internals. Add to this, it is difficult or impossible to merge your own patches into the mainline version of an OS kernel, especially when these patches are workload-specific. Moreover, note that despite kernels of most traditional OSes are extensible at runtime with kernel modules or kernel extensions, such runtime extensions do not allow extending the scheduler – which requires patching and recompilation of the entire kernel.

User space scheduling. Although built-in kernel scheduling policies are not reasonably modifiable, traditional OSes provide a rich interface for applications in user-space to change scheduling parameters and thread priorities, which can be used to create ad-hoc policies. For instance, by using a multi-queue FIFO scheduler (in kernel-space), where each queue is assigned a different priority, and playing with priorities (from user-space), other scheduling policies can be implemented, such as Earliest Deadline First (EDF). However, there are two main issues with this approach: first, in user-space, a scheduler has limited knowledge of the entire system; and second, implementing a scheduler in user-space may not be effective because of the additional overhead of context switches. Indeed, in order to know about the state of running applications and change their priorities, a scheduler implemented in user-space (with sufficient privileges) needs to invoke syscalls, which add considerable overhead to the scheduling itself. On top of this, certain information such as what thread holds which locks, may available in kernel-space (e.g., Futex), but not easy to export to user-space on traditional monolithic OSes.

Proposal. In this position paper we argue that it is time for OS kernel-level schedulers to be user-programmable, from at least a category of users, without any security related side-effects. We introduce our preliminary design that borrows the microkernels' design principle [22] of dividing mechanisms from policies, and applies that to monolithic OSes. All scheduling related mechanisms are always built-in in the OS kernel, while scheduling policies are modifiable, or definable, at runtime by users' applications (with specific privileges).

2 Motivation

Several recent papers [15, 16, 26] already highlighted the importance of user-programmable schedulers for several use case scenarios, including reducing network tail latency, reducing overheads of high-threaded bespoken applications, and improving the throughput of interdependent workloads. Interdependent workloads are extensively deployed in several scenarios, and increasingly in data-centers, including microservices, I/O-bound applications, parallel multiprocessing (e.g., MapReduce or MPIs) jobs, machine learning training jobs, etc., and are deployed as PaaS, IaaS, and FaaS.

A data-center scenario. Herein, we focus on FaaS-based microservices, which consist of several functions that are called in sequence or graph [1, 18]. FaaS platforms mainly involve the scheduling of code functions (e.g., Amazon Lambdas), being executed in functions' runtimes, based on events. When

faced with several sequences (or chains) of functions on a single server, especially on an overcommitted server, scheduling becomes tricky. We identified that functions' runtimes are executed even if their hosted functions have no work to do – i.e., there is no request to be handled,wasting CPU resources and slowing down other functions, with VM or containers.

Experimental Setup. We deployed Amazon Firecracker[1] on `a1.metal` servers on AWS EC2. We chose a chain of 5 microservices based on [10]. The chain performs 5 image processing functions from ServerlessBench [33] to generate a thumbnail. We launch the chain and record execution metrics, then repeat the experiment by increasing the number of colocated chains (from 0 to 50). Input and output images are read and store from-to AWS S3.

Experimental Results. On a single chain invocation, we observe that the unit's CPU idle time ratio ranges between 16% to 27% of the total CPU time used by the application. These numbers worsen to reach 69.25% when we increase the overcommitment ratio, thus undermining the execution times of functions. Idle times are curtailed when running the chains inside containers – since containers are processes, the OS scheduler has more insight on what threads are doing. We run our experiments a second time using Apache OpenWhisk [27] in standalone mode (v1.0.0), an open source FaaS platform that leverages Docker containers to run functions and observe container idle time ratios ranging from 17.8% to 31.6%, thus a lower CPU time waste ratio as compared to micro-VMs but still the issue remains.

Takeaway. User-programmable schedulers can decide to schedule function runtimes based on the knowledge of what event a function is waiting for – which is known to the cloud provider, to save CPU cycles.

3 Historical Perspective on Extensibility

Unsafe OS Extensibility. Traditional OS kernels are based on the monolithic OS design, including Linux and FreeBSD. While several other designs have been proposed [2, 11, 23, 30], monolithic remains the design of choice mostly for performance reasons. One of the main issues of monolithic kernels was runtime extensibility. Following the publication of several research works that targeted this issue [28, 29], OS architects solved that problem by introducing kernel modules (e.g., Linux) or kernel extensions (e.g., Darwin) that enable part of an OS, like a device driver, to be loaded while the kernel is running. Note that microkernels allow extension as a fundamental design choice: numerous kernel services run in user-space. Despite that, none of today's traditional OS kernels allow runtime extensibility of the in-kernel scheduler. Hence, notable projects in the realtime community [5, 7, 8, 32] tried to provide such functionalities via modules/extensions, but none made it to into the mainline.

Safe OS Extensiblity. An alternative and more recent method to enhance the functionality of an OS kernel at runtime, but in a controlled way, is eBPF. eBPF [14] is the successor of BPF [25], originally introduced for network packet filtering/statistics into the BSD OS kernel. eBPF is a restricted ISA, which can be easily verified for a set of properties and translated into native code for execution in kernel-space. eBPF code can only call a subset of functions, and access specific areas of memory, such as its context, and memory maps. eBPF code can be injected only by root or administrator users. An increasing number of Linux kernel subsystems are beginning to be extended with eBPF today [3, 12, 17, 31] and recent research works leverage the latter to improve kernel subsystems such as the network stack [15, 16].

4 User-programmable Schedulers

We believe that the time has come to make *OS kernel schedulers programmable at runtime, by a subset of users, in a controlled and safe way.* We propose injecting scheduling algorithms in the kernel at runtime, in such a way that its code can be checked for properties and access only the information and the function calls in the kernel that it is allowed to – since the code is exposed to sensitive data when it runs inside the kernel. The core idea is to keep existent OS kernel interfaces (syscalls), but extend them when necessary, while introducing a set of new interfaces for the controlled execution of the in-kernel scheduling algorithms.

Scheduling solely in-kernel. Our design targets traditional monolithic OSes, but we believe the same design also applies to other OS architectures, including microkernels and exokernels [11]. We propose an architecture in which there is an OS scheduler running solely in kernel-space, extendable at runtime by a subset of users of the system – not exclusively root, and extensions are moved into the kernel and compiled in a format that can be checked to assure integrity and security. In a virtualization setup we do not aim at substituting the guest OS scheduler [9, 19], instead, we aim at sharing information between the guest and host schedulers to improve scheduling.

Multiple algorithms and users. In our architecture, a core OS scheduler infrastructure manages multiple scheduling algorithms, ordered in a chain, similarly to Linux scheduling classes, and enables users with specific capabilities to add scheduling algorithms at runtime. A scheduling algorithm is a collection of methods and fields (variables), that can manipulate threads and/or process descriptors of an OS. We believe that having only a root user being able to configure the scheduler is too limited, but we do not want any user to be capable of doing that either. Therefore, we propose to add to the OS a set of capabilities related to scheduling.

Scheduler's sandbox. The scheduler is a core part of an OS, and inserting a buggy scheduling algorithm may easily render the entire system unusable. Moreover, an in-kernel OS scheduler may potentially access numerous user information, if not all. We aim to solve these potential problems by enforcing that the user moves its code into the kernel in a format that can be syntactically checked for specific properties or formally verified, and by limiting the exposed available interfaces based on the capability of the user that adds the scheduler algorithm. Ultimately, a scheduler algorithm code can be for example in an intermediate representation/bytecode and interpreted or JITed, but potentially similar properties can be guaranteed in another way (e.g., hardware-sandbox execution, and proofs on machine instruction code).

5 Conclusion

We highlighted that traditional OS schedulers, developed to support the general case, can be suboptimal, if not buggy, for certain workloads. Thus, users are looking for programmable schedulers. We presented a design and details on a future prototype to extend in-kernel OS schedulers at runtime with new policies, in a safe and controlled manner, by a group of users – not just root. Thus, usable also in cloud deployments.

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] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 267–283. https://doi.org/10.1145/224056.224077

[3] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. 2018. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. 108–110.

[4] 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

[5] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. 2006. Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 111–126.

[6] Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. 2019. Fork/wait and multicore frequency scaling: a generational clash. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. 53–59.

[7] Felipe Cerqueira and Björn Brandenburg. 2013. A comparison of scheduling latency in linux, preempt-rt, and litmus rt. In *9th Annual workshop on operating systems platforms for embedded real-time applications*. SYSGO AG, 19–29.

[8] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. 2011. ChronOS Linux: a best-effort real-time multiprocessor Linux kernel. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 474–479.

[9] 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)*. Article 12, 10 pages.

[10] 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]

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[12] 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

[13] Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. 2020. Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 435–448.

[14] Brendan Gregg. 2016. Linux 4. X Tracing Tools: Using BPF Superpowers. (2016).

[15] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast amp; Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[16] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 605–620. https://doi.org/10.1145/3477132.3483548

[17] Jakub Kicinski and Nicolaas Viljoen. 2016. eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev* 1 (2016).

[18] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. 2015. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2785–2792.

[19] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. 2012. Flattening Hierarchical Scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software* (Tampere, Finland) *(EMSOFT '12)*. Association for Computing Machinery, New York, NY, USA, 93–102. https://doi.org/10.1145/2380356.2380376

[20] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. 2022. OS Scheduling with Nest: Keeping Threads Close Together on Warm Cores. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New

York, NY, USA, to appear.

[21] 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

[22] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. 1975. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles* (Austin, Texas, USA) *(SOSP '75)*. Association for Computing Machinery, New York, NY, USA, 132–140. https://doi.org/10.1145/800213.806531

[23] Jochen Liedtke. 1995. On micro-kernel construction. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 237–250.

[24] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[25] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (San Diego, California) *(USENIX'93)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1267303.1267305

[26] Djob Mvondo, Antonio Barbalace, Alain Tchana, and Gilles Muller. 2021. *Tell Me When You Are Sleepy and What May Wake You Up!* Association for Computing Machinery, New York, NY, USA, 562–569. https://doi.org/10.1145/3472883.3487013

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

[28] Amit Purohit, Charles P Wright, Joseph Spadavecchia, Erez Zadok, et al. 2003. Cosy: Develop in User-Land, Run in Kernel-Mode.. In *HotOS*. 109–114.

[29] Margo Seltzer and Christopher Small. 1997. Self-monitoring and self-adapting operating systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE, 124–129.

[30] Christopher Small and Margo Seltzer. 1996. A Comparison of OS Extension Technologies. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (San Diego, CA) *(ATEC '96)*. USENIX Association, USA, 4.

[31] Viet-Hoang Tran and Olivier Bonaventure. 2019. Making the Linux TCP stack more extensible with eBPF. In *Proc. of the Netdev 0x13, Technical Conference on Linux Networking*.

[32] Victor Yodaiken et al. 1999. The rtlinux manifesto. In *Proc. of the 5th Linux Expo*.

[33] 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