Research Statement

Kostis Kaffes kkaffes@stanford.edu | web.stanford.edu/~kkaffes

My research interests span the areas of operating systems, cloud computing, and networking. The overarching goal of my research is to **democratize computer system development** so that their core components can be optimized for the benefit of individual users (i.e., developers) and their applications. My Ph.D. research has **given control over scheduling back to the users** and has **improved application performance**, **infrastructure efficiency**, and **scalability**. To achieve that, I have used techniques across the computing stack, ranging from developing algorithmic policies to implementing novel software mechanisms and frameworks to designing new hardware.

Different points in the system stack have different knowledge, permission, and deployment entry thresholds required to make contributions. For example, deploying a new serverless function might only require knowledge of a high-level programming language and platform-specific APIs, while developing new hardware requires costly simulations, verification, and fabrication. Naturally, most programmers stick to areas with lower thresholds missing the opportunity to customize the underlying computing stack for their needs. Even experienced programmers benefit from lower entry thresholds because lower thresholds shorten the development-test and deployment-deploy cycles. However, some of the most critical system components, such as schedulers, have some of the highest entry thresholds. Scheduling determines how the system resources, e.g., cores, network bandwidth, etc., are partitioned across and within applications and can impact performance and scalability by orders of magnitude. The entry threshold for scheduling is particularly high; it can span across multiple layers of the stack, ranging from the network interface card to kernel modules to userspace, while interacting with various low-level mechanisms and the hardware.

I led the quest to lower the entry threshold for scheduler development by addressing the following two key questions:

1. *What are the performance benefits of custom scheduling?* The first step of my research methodology is to provide a proof of concept and measure the potential benefits of my approach. To do so, I focused on two emerging application types that are increasingly important in computing today, microsecond-scale workloads and serverless functions. First, I developed Shinjuku [1] (NSDI 2019), a dataplane operating system that transfers control over scheduling from fixed-function hardware to software by enabling preemption at microsecond scale. Using Shinjuku, I showed that better scheduling improves end-to-end application performance up to 8× for microsecond-scale workloads. Second, I implemented a new scheduler for serverless functions [2], the dominant emerging computing paradigm, that caters to their unique characteristics, improving performance by up to 6.6× compared to existing schedulers.

2. *What are the right abstractions and mechanisms that make scheduling easily customizable throughout the stack?* The next natural question to answer was how to lower the entry threshold for scheduling customization even further and make it more accessible by non-expert users. To that end, I developed Syrup [3] (SOSP 2021), a framework that allows application developers to easily specify custom scheduling policies as matching functions in just a few lines of code and safely deploy them across the stack. Moreover, I am currently working on DBOS [4] (VLDB 2022), a database-backed cluster-level operating system that enables the implementation of core system functionalities in high-level declarative code.

Throughout my Ph.D., I aimed to do impactful work. As a result, I have published the results of my research in top systems, networking and database conferences (including SOSP, OSDI, NSDI, and VLDB), released open-source implementations, and, as I describe throughout my statement, had my ideas adopted by big technology companies.

1 Performance Benefits of Custom Request Scheduling

Microsecond-scale Workloads. The need for low tail latency in modern distributed systems has led to the development of specialized dataplanes that bypass the Linux kernel and deliver networked requests directly to applications. Such systems use fixed hardware-implemented policies to minimize overheads when scheduling remote procedure calls (RPCs) to cores running application code. Moreover, cores process requests in a First-Come-First-Serve fashion as preemptive scheduling was considered incompatible with microsecond-scale latency targets. Unfortunately, these fixed policies are sub-optimal for broad classes of workloads, e.g., ones where request service times follow distributions with high dispersion or a heavy tail, limiting single-server performance and driving an under-utilization crisis. Even big technology companies that can compensate through top-quality engineering effort and custom networking hardware suffer from these problems, let alone the rest of the world that lacks such resources.

To solve this problem, I developed Shinjuku [1], a dataplane operating system that radically upends the notion that preemption cannot be used in low-latency systems and lowers the entry threshold for request scheduling from hardware to software. The key idea behind its design is the use of hardware extensions for virtualization for the generation and delivery of interrupts with very low overhead, enabling preemption at microsecond scale in software. This mechanism allows Shinjuku to implement centralized scheduling policies that preempt requests as often as every 5μsec and work well for light and heavy-tailed request service time distributions. For example, when used to schedule RocksDB requests, a database developed and extensively used by Facebook, Shinjuku achieves up to 6.6× higher throughput and 88% lower tail latency over state-of-the-art dataplane operating systems. Shinjuku is considered the state-of-the-art scheduling system for highly variable workloads and is used as a baseline by several other systems. Moreover, the centralized scheduling in Shinjuku influenced the design of a custom scheduler used internally by Google for search workloads.

In follow-up work, we expanded Shinjuku's key ideas across different dimensions. First, we showed that we can combine Shinjuku with emerging programmable networking switches to schedule requests at cluster scale [5] (OSDI 2020). Second, by taking advantage of programmable network devices, we moved Shinjuku's scheduling back to the hardware without raising the entry threshold [6] (HotNets 2019).

Serverless Functions. An emerging paradigm in cloud computing is to replace RPC calls to long-running servers with serverless functions. Function-as-a-service (FaaS) lowers the entry threshold for cloud application development as users simply need to implement a function at a high-level language and specify execution triggers. As a result, cloud providers now manage tasks previously handled by users such as resource provisioning, scheduling, scaling, and security. For the serverless programming model to succeed, providers must perform these tasks as optimally as the users do.

I focused on optimizing a task critical to FaaS performance, scheduling of serverless function invocations across and within machines [7] (SoCC 2019). Scheduling is particularly challenging in the serverless setting as function invocations are bursty, have short but highly variable execution times, and have high start-up costs making migrations uneconomical. These characteristics make commonly-used techniques such as late binding and random load-balancing sub-optimal. Starting from a clean slate, I designed Hermes [2], a new serverless function scheduler. Hermes handles high execution time variability by early binding function invocations to servers and letting them receive a slice of the processor as soon as they enter the system. It also uses a load-balancing approach that is load, locality, and cost-aware, packing invocations to servers with "warm" containers, i.e., containers already initialized with the function code, without overloading them and causing queuing delays. I integrated Hermes with OpenWhisk, the leading open-source serverless platform, and showed that it provides up to 85% lower slowdown and supports up to 60% higher load than existing approaches.

2 Custom Scheduling Across the Whole Stack

Syrup. Even when done in software, prototyping, evaluating, and deploying new scheduling policies is an arduous process. Due to the complexity of modern operating systems, developers who want to implement custom scheduling policies avoid hacking the kernel and develop new dataplanes from scratch, sacrificing compatibility and maintainability. Every year, there are multiple papers on custom schedulers published in top systems and networking conferences showcasing both the need for innovation and the high entry threshold for their development. The ultimate goal of my doctoral research was to lower this threshold and make it *easy for users to modify scheduling* and customize it to serve their applications better.

To achieve that, I developed Syrup [3] (SOSP 2021), a framework for user-defined scheduling. It enables users to express their scheduling preferences in a high-level language and deploy them on top of existing operating systems and across networking infrastructure. Untrusted application developers can use Syrup to define and safely deploy arbitrary scheduling policies across various layers of the stack, including the kernel networking stack, the kernel scheduler, and programmable network devices. Syrup makes specifying scheduling policies easy by treating scheduling as a matching problem. Users specify scheduling policies by implementing functions that match inputs, e.g., packets or threads, with executors, e.g., sockets or cores, without dealing with low-level system details such as how inputs are steered to the selected executors. The high-level policy code is automatically and safely integrated with the various scheduling mechanisms throughout a modern system. Syrup won a 2020 research award in networking from Facebook.

We can use Syrup to define application and workload-specific scheduling policies in a few lines of code, dramatically lowering the entry threshold for scheduling policy development. For example, we can implement the Size Interval Task Assignment (SITA) policy for RocksDB and improve performance up to 8× compared with the default Linux policy, delivering a significant fraction of the performance benefits possible with an application-specific operating or runtime system. A previous implementation of SITA published in NSDI required building a custom dataplane operating system from scratch and heavily modifying application code. Our implementation of SITA in Syrup is just 16 lines of code and requires no application modifications.

DBOS. Customization should not be limited to single-node scheduling operations. To achieve this goal, I am currently working on DBOS [4, 8] (VLDB 2022, CIDR 2022), a novel cluster operating system. DBOS stores state from all levels of the stack, from high-level applications down to core services like schedulers and file systems, centrally in a single distributed transactional database. This design allows the implementation of OS services such as scheduling, file management, and inter-process communication as standard database queries without sacrificing performance and dramatically reducing code complexity. For example, we can implement a FIFO task scheduler in 10 lines of SQL code and make it locality-aware with just one additional line.

Industry Outreach

I firmly believe that to tackle some of the greatest societal and technological challenges in computing today cooperation between industry and academia is necessary. Thus, in the past I have applied my research to industrial settings. While an intern at Google, I developed and deployed Per Application Class Turbo Controller (PACT) [9] (SoCC 2020), a framework that uses core scheduling and dynamic frequency scaling to reduce power consumption in highly utilized data centers without sacrificing application performance. PACT's main advantage is that it is application-agnostic. As a result, it does not require any application-specific performance signals and can thus be deployed at Google's datacenters lowering power consumption by 9% and reducing carbon emissions.

Looking forward, I plan to continue creating such synergy. For example, in looking for ways to further improve the infrastructure for scheduling, I recently proposed a new CPU hardware design with a radically different hardware threading model [10] (HotOS 2021). The new model adds a large number of hardware threads to each physical core – making software thread multiplexing unnecessary – and lets software manage them - making scheduling simpler. Our design has already drawn interest from the industry as a prominent chip manufacturer is interested in building a prototype chip.

Future Directions

There is still huge room for improvements in performance and efficiency in modern systems that are imperative to achieve to continue scaling in the post-Moore's Law era. The question now is how to define solid abstractions and implement scalable systems that lower the entry threshold for computer system development and foster innovation in different components across the stack.

Domain-specific abstractions for scheduling. Scheduling occurs across the data center stack, from cluster managers and software load balancers to programmable switches and machine learning accelerators. I plan to extend Syrup to support user-defined scheduling across such backends. Moreover, users are currently burdened with writing their scheduling policies in specialized languages for each target backend, e.g., P4 for programmable switches. My goal is to remove that burden from them by introducing a programming language that raises the level of abstraction for defining scheduling policies. We can build this language around the matching view of scheduling we used in Syrup and compile it for different backends, ranging from eBPF to P4 to high-level languages.

Customizing the stack, beyond just scheduling. There are also many single-host customization directions I am excited to explore. For example, custom memory management can enable applications to decide which pages to store in fast and which in slow memory at a fine scale. Similarly, congestion control protocols that take direct feedback from the application can avoid the buffer overflow associated with the slow receiver problem. Currently, developers can only do that by using hard-to-develop library operating systems that are often incompatible with existing applications. Thus, similar to what I did for scheduling, I am eager to make fundamental operations of widely-adopted operating systems such as Linux fully customizable and programmable by users in a secure and performant manner.

Self-driving systems. My work provides interfaces for cross-cutting customization of fundamental system operations. Now the question becomes who writes the customization program and how can we reach global optima. Expert developers will always be able to fine-tune the operating system stack and significantly improve application performance. However, my long-term goal is to provide such benefits even to everyday developers who do not want to deal with the operating system.

We can achieve this by making the operating system self-driving through the use of machine learning. By exploiting the novel APIs I introduced to systems and taking an end-to-end perspective, we can deploy operating system components that learn from the behavior of different applications and self-adapt to serve them better. Many exciting research questions in the systems and the ML side need to be answered before such a design becomes practical. Should machine learning be used to choose among a set of predetermined policies, or should the policy itself be a learned function? Can we deploy low latency models that operate at nanosecond-scale, e.g., at a per-packet level? How do we decide what to use as input features for each different system component? How do we quantify the effect of each OS component on the end-to-end performance?

Conclusion

Recent trends in software and hardware development have made the need for democratized systems particularly urgent. Big technology companies are creating walled gardens, i.e., closed ecosystems that integrate everything from hardware to software to distribution, exposing only high-level APIs to users. Such an environment stifles innovation. Users cannot observe what is happening under the hood, identify possibilities for improvement, and most importantly, adjust the systems' behavior to suit their applications better. However, by continuing innovating in an academic setting, I can help tear down these artificial barriers and make computer systems open and extensible, ushering in a new golden age of system and application development.

References

[1] **Kostis Kaffes**, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[2] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Practical Scheduling for Real-World Serverless Computing, November 2021. arXiv:2111.07226.

[3] **Kostis Kaffes**, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling across the Stack. In *Proceedings of the 28th Symposium on Operating Systems Principles*, SOSP '21. Association for Computing Machinery, October 2021.

[4] Athinagoras Skiadopoulos*, Qian Li*, Peter Kraft*, **Kostis Kaffes***, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A DBMS-oriented Operating System. *Proc. VLDB Endow.*, 15, 2022.

[5] Hang Zhu, **Kostis Kaffes**, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.

[6] Jack Tigar Humphries, **Kostis Kaffes**, David Mazières, and Christos Kozyrakis. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 60–68, Princeton, NJ, USA, 2019. Association for Computing Machinery.

[7] **Kostis Kaffes**, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized Core-Granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 158–164, Santa Cruz, CA, USA, 2019. Association for Computing Machinery.

[8] Qian Li*, Peter Kraft*, **Kostis Kaffes***, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Jason Li, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. A Progress Report on DBOS: A Database-oriented Operating System . In *Conference on Innovative Data Systems Research*, CIDR 2022, Chaminade, USA, 2022. CIDR.

[9] **Kostis Kaffes**, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. Leveraging Application Classes to Save Power in Highly-Utilized Data Centers. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 134–149, Virtual Event, USA, 2020. Association for Computing Machinery.

[10] Jack Tigar Humphries*, **Kostis Kaffes***, David Mazières, and Christos Kozyrakis. A Case against (Most) Context Switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 17–25, Virtual, USA, 2021. Association for Computing Machinery.