On the Impact of Isolation Costs on Locality-aware Cloud Scheduling

Ankit Bhardwaj Meghana Gupta Ryan Stutsman
University of Utah

Abstract

Serverless applications create an opportunity for more granular scheduling across machines in cloud platforms that can improve efficiency, especially if functions can be run within storage services to eliminate data movement. However, embedding code within storage services creates code isolation overheads that offset some of those savings. We argue for a new approach to serverless function scheduling that can look within serverless applications' functions, profile their data movement and networking costs, and model the impact of different code placement and isolation schemes for those costs. Beyond improvements in efficiency, such an approach would fuel innovation in cloud isolation schemes and programming abstractions, since a scheduler with a modular cost modeling approach could incorporate new schemes and automatically use them to improve efficiency for pre-existing applications.

1 Introduction

How cloud platforms allocate, schedule, and place computation is restricting the evolution of cloud computing. The last 5 years have seen an explosion in data center network performance, a breakdown and rethinking of CPU protection mechanisms, and a move of applications from coarse, opaque virtual machines (VMs) to granular serverless functions.

These three transformations have a complex interrelationship that could enable a new era of more efficient systems and rapid evolution of both cloud abstractions and mechanisms. We expose this relationship and show that explicit automated reasoning about data movement and CPU hardware isolation costs can improve efficiency in the cloud. In reaction, we propose a new approach to compute provisioning and scheduling in the cloud that uses the emerging visibility into (serverless) applications. It breaks down static disaggregation of compute and storage by reasoning explicitly and at fine-grain about trade-offs between data movement and code isolation costs to maximize efficiency. Ultimately, we argue that such an approach can accelerate innovation by automating reasoning about the costs and trade-offs of new network transport schemes, code isolation schemes, and applications interfaces, since this yields a "pluggable" platform that can incorporate and optimize around heterogeneous workloads, datasets, isolation models, and computation models.

2 Toward Granular Scheduling in the Cloud

Today, cloud providers avoid fixed allocations of compute (AWS EC2 and Lambda; Azure VMs and Cloud Functions) and storage resources (AWS S3, EBS, and DynamoDB; Azure Blob, Disk, and SQL) through disaggregation (Figure 1 top). Decoupling compute and storage resources lets any VM use any storage, improving utilization. However, disaggregation forces data movement over the network, which has a CPU cost and cuts into efficiency. Customer (or *tenant*) applications could hold storage data locally within VMs (Figure 1 middle), but this creates scaling challenges when more computation is needed than a single VM can supply. Cloud providers could also allow tenants to run VMs on the same machines that hold tenants' data (Figure 1 bottom), but this doesn't work because many tenants share a set of storage servers. Thousands of tenants' data are inter-mixed across many machines to homogenize load, but running thousands of different tenants' VMs on a single machine is prohibitively inefficient. Effective efficiency is, then, determined by two things: the cost of moving data over the network and the cost of consolidating tenants' storage and compute workloads on a set of machines.

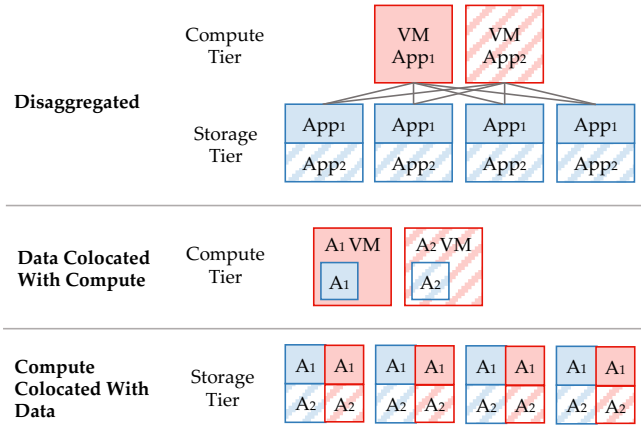

Figure 1: Possible placements of code and data in the cloud.

Aside from heavy isolation costs, VMs are also problematic because they are coarse and only offer a handful of placement choices relative to the data they operate on (all shown in Figure 1); they are opaque, so it is hard to assess what they compute, what their inputs/outputs are and how they interact with data. However, serverless functions have begun to change this. Serverless has shown that developers are willing to decompose and repackage applications with new abstractions that move beyond the legacy POSIX interfaces of the past several decades.

Serverless opens up several new opportunities. First, it shatters an application from a monolithic VM into tens or hundreds of relatively independent pieces of computation that can all be *individually placed*. Second, it *creates visibility*

into applications; functions' inputs, outputs, and data interactions can all be observed and attributed to granular application pieces, creating opportunities for more intelligent placement. Third, the new interfaces of serverless *support alternate isolation schemes* that are less expensive than conventional VMs. Today, cloud providers don't exploit this flexibility, but it is an opportunity for innovation. Finally, the interface is new enough that it is *malleable*. Serverless has shown that developers are willing to adopt new abstractions if those abstractions ease development, deployment, and scaling. This makes the case for exploring new abstractions for computation in the cloud that are co-designed for ease of development *and* lowering tenant code isolation costs.

These four opportunities could all be exploited today, but today's serverless platforms, instead, favor backward compatibility and classic hardware virtualization techniques for isolation [1]. A key problem is that no single "tweak" to the existing model can instantly and easily provide a massive efficiency benefit warranting complete rework of cloud providers' platforms and tenants' existing codebases. This is compounded by the fact that each small change to cloud provider isolation mechanisms or compute abstractions requires a ground-up rethinking of all costs and trade-offs. For example, if a provider created a new low-cost abstraction for embedding short functions within storage servers (for example, via eBPF [5]), then data movement costs, isolation costs, and function granularities would all need to be reconsidered for this change in order to ensure it improves efficiency. Hence, the missing piece that would enable faster innovation is fully automating the reasoning needed to incorporate new abstractions for computation, new schemes for efficient data movement, and new schemes for low-cost code containment.

Our assertion is that it is time for a more principled approach to reasoning about the trade-off between data movement and code isolation costs. This paper's aim is to outline a new distributed scheduling layer for cloud platforms that transparently optimizes tenant code and data placement; such a layer could automatically execute a tenant's function code directly on a storage server when it makes sense just as it might disaggregate when isolation code costs are too great.

3 Challenges in Finer-grained Scheduling

Though serverless platforms open up new possibilities in scheduling compute resources to lower data movement overheads, there are several key challenges that need to be addressed to support this type of optimization. Here, we discuss these challenges, including application decomposition, workload characterization, provisioning schemes, isolation mechanism costs, and function intermediate state movement. In the following section, we outline a design that explicitly considers these issues to try to minimize cloud application data movement and isolation costs.

Granular Application Decomposition. To schedule and gain visibility at a fine-grain, applications must be specified at a fine-grain. Developers already do this today to gain the benefits of serverless computing. Serverless applications are comprised of a set of functions invoked by a set of triggers (e.g. external HTTP requests, timers, internal cloud service/storage triggers, etc.). Applications are scheduled and scaled across cloud hardware resources on demand; subsequent and proximate events for functions within an application can reuse existing application instances [27].

In the future, applications can be augmented in some simple ways that will give a scheduler more flexibility in minimizing data movement and cross-function communication/invocation costs. For example, function inter-dependencies are not specified upfront and are difficult to extract. Allowing applications to specify or give hints about their structure and chaining (for example, as a graph of dependencies between functions and external services) could allow a scheduler to run functions next to the data they plan to operate on. It could also be used to co-schedule functions to minimize context switch and communication overheads across functions within an application.
Requirement 0: Granular Visibility and Placement. Today's serverless platforms are already driving developers to expose application components to cloud providers, creating the opportunity for better scheduling through better understanding of their behavior, communication, and resource needs.

Workload Characterization. Colocating functions with the data they operate on always reduces data movement, but the benefit of this colocation depends on two things. First, recent reports suggests overhead due to remote accesses occupy 22-27% of fleet-wide CPU at Google [11] with Facebook reporting similar numbers [30], but the cost for moving data over a network differs for different storage platforms and networks. For example, an application with low spatial locality may spread fine-grained requests for small data records across a large set of servers; such a workload is primarily dominated by per-request CPU costs, like request dispatching costs. Other workloads may focus storage requests for larger items onto a smaller set of servers, making CPU-cost-per-byte the dominant cost factor. Complicating things, even within a single cloud platform, many forms of networking exist all with different costs. For example, one class of Azure VM has an unaccelerated networking configuration and at least three other forms of accelerated networking [20, 21]. Hence, to minimize costs in placement a scheduler needs a detailed, runtime-calibrated cost model to assess data movement costs.

Second, each cloud application consumes CPU time and interacts with storage in a unique way. Even within one application, some parts of an application may benefit from execution near storage, whereas other parts may create CPU bottlenecks at storage. Figure 2 shows this; it shows the execution of a serverless-style function executing against a prototype storage system that uses accelerated networking and a low-cost function runtime [14]. When the function accesses more than one data record in the store, storage server throughput is maximized when the function is run *on* the storage server since

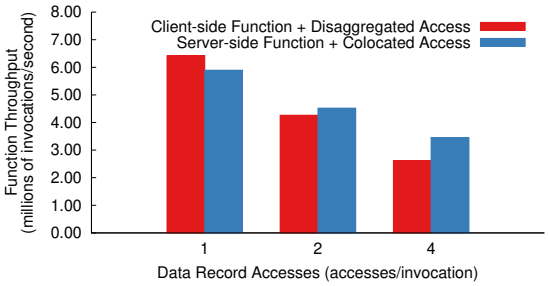

Figure 2: Throughput of a storage server (a prototype with accelerated networking and a specialized, low-cost code isolation scheme) when a serverless function is run inside and outside the server. When functions access multiple values with a single remote function invocation, running functions at the server can improve storage server throughput by reducing networking and dispatch costs.

it reduces network transport and request dispatch overheads. However, this is only true if the function does not contain computationally-intensive logic. Serverless helps by giving schedulers a more granular view of an application, creating opportunities to place parts of an application within storage only if they eliminate enough networking CPU cost at storage to *improve* storage server throughput. However, to do this, the scheduler needs more than just a network cost model; it must be able to profile and assess how each function invocation interacts with storage and uses CPU.

Requirement 1: Automated Workload Profiling. A scheduler that can optimize placement of cloud functions near stored data would need to build a network cost model and to internally profile data access behavior within functions. Simple schemes we have tried purely using runtime measurement show some gains, but more advanced modeling and prediction of the impact of function inputs on outputs and data access rates, sizes, and burstiness could be used to further optimize function placement.

Context Switch Costs. Each different scheme for safely isolating and executing code and each different workload has different costs when a code execution context must be scheduled or de-scheduled. For example, a cloud tenant making requests to a serverless cloud function at a low rate will either force a CPU to remain idle in the cloud awaiting those requests, or it will force a form of context switch to handle those requests, increasing the effective CPU cost of each request. At low request rates one isolation scheme may be more efficient, and at higher requests rates another isolation scheme may be more efficient.

Hence, where a particular function should be run depends not just on data movement and network CPU costs, but also on the "temporal locality" of requests coming from that tenant. That is, the "hit rate" of processing requests on a pre-existing, active context where no context switch is needed and the cost of context switch to schedule a context to handle the request when there is no active context must be considered when scheduling. Hence, to optimize, a scheduler needs to have a global view of active compute contexts, and it needs to be able to reason about the costs of dispatching to those contexts.

Beyond temporal locality, different protection schemes have different context switch costs. Today's serverless runtimes support POSIX, but practical serverless applications rarely need to run native code using legacy I/O interfaces. This creates a space where static analysis may be able to group functions into those that can run safely in lighter-weight runtimes (language-level or non-VM hardware-based isolation schemes) and those that need backward compatibility.

Requirement 2: Global, Activation- and Isolation-cost-aware Scheduling. Minimizing data movement and network costs is insufficient alone if function placement results in idle CPUs or extra context switch costs per network request. Function scheduling must maintain a fine-grained view of where function contexts are scheduled in order to reduce context switch costs between protection domains. This is a major challenge at scale, where requests and context switch can complete in a few microseconds. The scheduler must also be aware of application quality-of-service needs to understand where the scheduler can leverage request batching to amortize context switch and dispatching costs and where fast response times are needed by applications.

(Re)provisioning and Placement. Trading off data movement costs of different placements of function invocations relative to data creates an NP-hard, large-scale bin packing problem. Tenants with different resource requirements (low CPU cost/high network use, high CPU cost/low network use) should be placed together to improve utilization, subject to each function's characterized costs and data access patterns.

Complicating things, the efficiency of function isolation schemes (VMs, containers, specialized hardware or software runtimes, etc.) depends on several properties of the function being contained. Specifically, for functions with high, predictable, and stable CPU use and data access patterns, VMs with dedicated hardware resources can be most cost effective. However, VMs have heavy-weight environments that make them costly to start, stop, and migrate, even for applications with (even predictably) changing resource requirements; thus, they leave hardware under-utilized. While placing a VM within a storage server might be efficient in certain cases, the disruption and resource use of migration needs to be considered [13]. This makes lighter-weight runtimes like those possible for serverless platforms attractive for automatic placement tuning, especially when load is unpredictable.

Requirement 3: Explicit Workload Stationarity Modeling. VMs are efficient when workloads are stable but costly under change. Serverless runtimes pay extra software dispatch overheads, but scale more readily. Finally, creating/destroying VMs or containers requires some initialization time and cost along with some resources. An efficient cluster scheduler should reason about the predictability of a workload and about the cost of moving the computational environment and data of applications' functions that it is placing for execution.

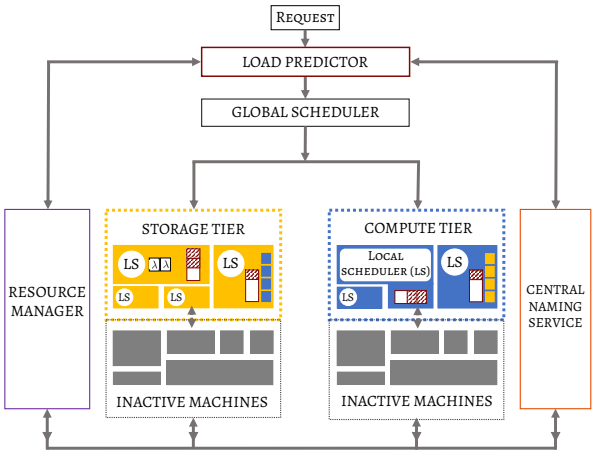

Figure 3: Sandstorm Architecture.

Intermediate State. Recent serverless applications have mostly been stateless, where invocations of the same function do not share state. VMs have historically run tenant-specific logic, so coordinated access to shared state (e.g. held in VM memory) by different requests is common. Intermediate state that applications build up complicates placement decisions if consistency guarantees are made across shared state. Synchronizing intermediate state and reasoning about the costs of synchronizing it would complicate optimization of placement. However, today's serverless models don't support this (well).

Requirement 4: Stateless and Motionless Invocations. Stateless functions let each invocation of a function be placed separately without concern for coordinating with existing invocations of the same function. This makes optimization easier since placement can be decided invocation-by-invocation, and it effectively eliminates the need for state migration. Functions deemed to benefit from a new form of isolation or new placement can have new invocations run in the new configuration, letting existing invocations complete in their original configuration. Our goal is to support an approximation of statefulness by moving functions that keep intermediate data near to the (function external) physical location where they store that data rather than building a new, specialized consistency scheme specifically for intermediate function state.

4 A Preliminary Design

To explore our idea, we are designing a new serverless scheduler we call Sandstorm that is tightly-coupled with the machines it schedules over. In most ways, Sandstorm is like other serverless schedulers (Figure 3). It includes a load balancer/scheduler that makes machine allocation and function placement decisions and a scheme for collecting fleet-wide metrics to assist in scheduling. Like some other works [16,28], it is designed to run serverless function chains and can take into account per-function-chain-invocation hints that statically describe how functions in a particular chain are interconnected with one another. This lets it fuse or coschedule operations, and it lets the scheduler know when and where embedding the function within a storage service machine might make sense.

One benefit of this approach is that cloud providers can transparently improve their utilization of compute, storage, and network infrastructure. This would allow them to consolidate more tenants on fewer machines while retaining their current pricing models, which only account for CPU and memory allocated to functions for the duration of their execution [3].

A key idea in Sandstorm is that it tries to collapse function chains into the storage tier to control data movement. However, as we have laid out, done naively this could hurt efficiency and create bottlenecks. This has led us to a design centered on aggressive collection of low-level performance data to calibrate network and isolation costs to predict load stability; to maintain a sub-millisecond-scale view of the status of each core across all machines; and to inform global sub-millisecond remapping of cores/redirection of incoming tenant requests among cores fleet-wide. This would let Sandstorm promote and demote cores at scale in a few milliseconds between several regimes of operation. For example, it can remap *any* core in *any* compute-tier or storage-tier machine to run a function on dedicated CPUs with dedicated hardware network queues (via SR-IOV); or with fast, lightweight temporal multiplexing of functions designed for software dispatching and low context switch cost; or by moving execution of the function out of a storage-tier machine to a compute-tier machine. The last option would force remote storage access to lower code containment costs or to gain access to idle compute resources. We detail some challenges and mechanisms needed to support this kind of global optimization below.

4.1 Storage & Internal Task Dispatching

Sandstorm is unique in executing tenant-provided functions within storage-tier servers, and this directly impacts its server-local task dispatching. Figure 4 shows how this might work. Internally, storage-tier servers efficiently service requests for tenant data by polling queues that are directly filled by network card (NIC); similar to other kernel-bypass-based systems, this lets the NIC assist in dispatching, avoiding cross-core coordination and improving server throughput. All tenant functions can access any in-memory tenant state directly through shared memory, but data on other media can be accessed through runtime-provided interfaces. There is no predefined split between cores used for running tenant functions and simple read/write/get/put requests for stored data; servers profile offered load and dynamically re-provision cores between tenant logic and storage request processing as needed.

For tenants that only run functions at storage infrequently, the system must be able to switch between isolation contexts at low cost. Different isolation schemes have different tradeoffs, but, for example, for certain restricted functions, fast user-level protection schemes based on new hardware functionality like Intel's MPK [7,9,24,31] can efficiently switch between many domains on a single core even if functions

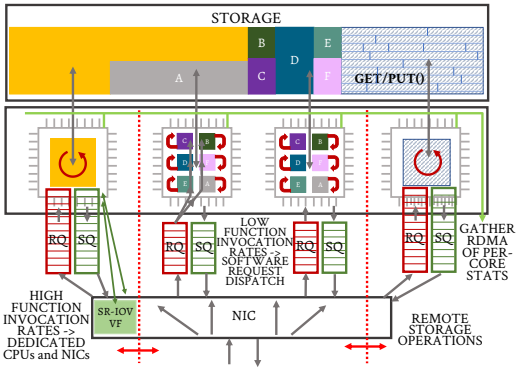

Figure 4: Core-level dispatching in a Sandstorm storage server.

are short-running and invoked millions of times per second in aggregate on a CPU core. For tenants sending the heaviest request rates to a server, it may make sense to dedicate whole cores with their own NIC hardware queues, avoiding all dispatch and context switch costs. In cases where the storage server becomes compute congested, this information propagates back to the scheduler and some fraction of tenant functions are redirected to idle compute capacity elsewhere in the the cluster. In this case, detailed data movement cost models and a function-level understanding of data access patterns is key, since this results in more data movement over the network. Hence, this should only be done when it wouldn't further increase the storage server's load.

4.2 Statistics, Scheduling, & Load Prediction

To make fine-grained scheduling decisions, tenants would invoke functions through a scheduler handling a large partition of tenants. For each invocation, the scheduler decides placement, routing it to the correct core in the correct machine. This placement is based on idle compute capacity; function data access patterns; and networking, dispatch, and isolation domain context switch costs. A tightly integrated naming service tracks all pre-existing execution contexts for a given function across the fleet along with placement information for any data owned by that tenant. The scheduler looks for a cost effective place to run a particular invocation and forwards the invocation request to the correct core in the fleet.

Another service predicts the stability of a function's behavior via history of its storage and network interactions and CPU cost breakdowns of those interactions. Consistently "hot" functions will trigger allocation of whole CPU cores and hardware NIC queues for future invocations; functions that suffer temporary load spikes can rely on software dispatching without triggering allocation of dedicated hardware resources.

Load prediction, naming, and cost estimation require fine-grained core- and task-level visibility — providing this without creating bottlenecks and high CPU load is a challenge in itself. Doing this efficiently will require hardware-assisted metadata aggregation. One possibility is to have each machine use gather DMA to collect per-core statistics with periodic RDMA writes that push those statistics into a pre-indexed structures within the scheduler machines. Even with more than 1,000 machines pushing statistics and metadata each millisecond a single scheduler machine would be sufficient.

5 Related Work

Serverless Frameworks. Since the advent of Amazon AWS 15 years ago [4], serverless frameworks have created a new scheme for specifying and scheduling cloud applications [3, 6, 19]. Similar open source and research frameworks have emerged [8, 29], but they largely keep similar, cost-agnostic scheduling policies to their industrial counterparts.

Memory Protection and Isolation. Recent work has explored new hardware-based memory isolation using MPK [7, 9, 24, 31]. Both recent research and industry efforts have explored lighter-weight schemes for VMs and containers, especially for serverless [1, 2, 15, 18, 22], and some works specifically address isolation schemes for computation embedded near stored data [14, 26, 32]. Sandstorm avoids focusing on specific memory isolation techniques; its goal is to separate mechanism and policy by trying to incorporate new mechanisms through an extensible cost model.

Scheduling. A great deal of research has focused on distributed scheduling for coarse-grained tasks like those in data parallel computational analytics frameworks. Recent works have focused on request scheduling in online applications with high variance in per-request computational costs, for example, to optimize latency/throughput trade-offs [10, 23, 25]. Sandstorm proposes extending these approaches to account for inter-tenant code isolation and data movement costs in distributed task scheduling.

6 Conclusion

Disaggregation drives cloud utilization today, but as Dennard scaling ends, real efficiency will be increasingly important. Future cloud platforms will need to exploit the newfound flexibility and granularity in how (serverless) applications are specified and scheduled to reduce data movement. However, this is difficult to get right: shipping all code to data creates additional costs for code isolation, scheduling, and dispatching that can more than offset the savings. In response, we offer new ideas on how a new cost-driven approach to scheduling compute resources can accelerate innovation in cloud isolation schemes and programming models by letting cloud platforms readily incorporate new schemes.

Acknowledgments

Thanks to our reviewers and our shepherd, Adam Belay, for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by Facebook and VMware.

Discussion Topics

Our single machine in-memory store prototype shows gains in throughput and latency when we move compute to storage; however, showing similar benefits at scale is harder. Here are some of the open problems and reservations we have about the idea for which we seek input from the research community.

Client-side Caching. Caching reduces data movement costs. Providers may get similar gains using client-side caching instead of moving compute to storage. Caching behaves poorly for write-intensive workloads where most functions modify data. Overall, the gain due caching depends on the workload and data consistency model. More knowledge about cloud-providers' workloads would help us in deciding if the caching is sufficient and for which classes of workloads co-locating computation and data dominates caching.

Process Model. Dynamically placing a binary at different locations at runtime requires small program/environment sizes and limited/unified interfaces for accessing external resources. Designing a new interface that is sufficient for serverless functions is an open problem. Furthermore, standardizing execution environments for functions so that large classes of them can be executed in a small number of environments is a challenge (to prevent proliferation of one-off environments that would restrict scheduling due to high environment initialization costs). Discussion on what types of interfaces are likely to emerge and are likely to succeed for developing real applications would be a helpful point of feedback.

Security risks. Many schemes can provide program or function isolation; however, verifying these schemes is hard, especially when we cannot guarantee that hardware itself is free from vulnerabilities [12, 17]. Sandstorm's adaptive placement and adaptive protection model scheme would multiply cloud providers' attack surfaces. A key question for discussion is what software and hardware isolation schemes are likely to be trustworthy enough to depend on in realistic, secure cloud provider platforms. For example, would verified JIT runtimes like eBPF be (or eventually be) secure enough to trust with containing tenant code? Relatedly, are microarchitectural mitigations for speculative execution attacks here to stay, and what impact will they have on the costs of (lightweight) isolation schemes? If flushes of microarchitectural state dominate isolation domain switch costs, how much improvement could different isolation schemes provide?

Workloads. Cloud platform research is hard with little visibility into provider infrastructure and workloads. We are always looking for input on how to validate ideas and systems like this, including feedback on what workloads to test with and where to find interesting workload traces.

Pricing. There is little public information about how cloud providers decide the parameters of pricing models used to charge tenants. Insight into pricing model strategies would help in understanding how specific changes in cloud internals might affect tenants' costs.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[3] Amazon, LLC. AWS Lambda - Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/.

[4] Amazon, LLC. AWS News Blog, Aug 2006. https://aws.amazon.com/blogs/aws/amazon_ec2_beta/.

[5] Matt Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[6] Google, LLC. Google Cloud. https://cloud.google.com/.

[7] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 489–504. USENIX Association, 2019.

[8] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[9] Intel, Inc. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[10] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 345–359, USA, 2019. USENIX Association.

[11] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[13] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Data Migration for Low-latency In-memory Storage. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*, SOSP '17, October 2017.

[14] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, Carlsbad, CA, October 2018. USENIX Association.

[15] James Larisch, James Mickens, and Eddie Kohler. Alto: Lightweight VMs Using Virtualization-Aware Managed Runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–7, 2018.

[16] Collin Lee and John K. Ousterhout. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 149–154. ACM, 2019.

[17] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[18] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[19] Microsoft, Inc. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[20] Microsoft, Inc. Create a Linux virtual machine with Accelerated Networking using Azure CLI. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli.

[21] Microsoft, Inc. Enable InfiniBand with SR-IOV. https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/hpc/enable-infiniband.

[22] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[23] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[24] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 241–254. USENIX Association, 2019.

[25] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 325–341. ACM, 2017.

[26] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 175–190. ACM, 2017.

[27] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider, 2020.

[28] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A Scalable Low-Latency Serverless Platform. *arXiv preprint arXiv:1911.09849*, 2019.

[29] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[30] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 513–526, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-Process Isolation with Memory Protection Keys. In *Proceedings of USENIX Security Symposium*, 2019.

[32] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.