Theme Article: Top Picks

Unveiling the Hardware and Software Implications of Microservices in Cloud and Edge Systems

Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou
Cornell University

***Abstract*—Cloud services progressively shift from monolithic applications to complex graphs of loosely-coupled microservices. This article aims at understanding the implications microservices have across the system stack, from hardware acceleration and server design, to operating systems and networking, cluster management, and programming frameworks. Toward this effort, we have designed an open-sourced DeathstarBench, a benchmark suite for interactive microservices that is both representative and extensible.**

Cloud computing now powers applications from every domain of human endeavor, which require ever improving performance, responsiveness, and scalability.[2,5,6,8] Many of these applications are *interactive*, *latency critical* services that must meet strict performance (throughput and tail latency), and availability constraints, while also handling frequent software updates.[4–7,12] The past five years have seen a significant shift in the way cloud services are designed, from large monolithic implementations, where the entire functionality of a

Digital Object Identifier 10.1109/MM.2020.2985960
Date of publication 22 April 2020; date of current version 22 May 2020.

Published by the IEEE Computer Society

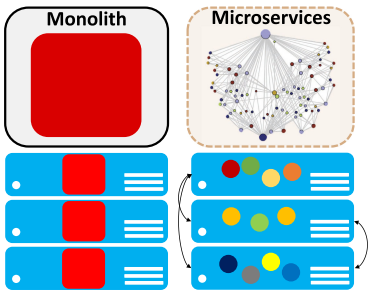

Figure 1. Differences in the deployment of monoliths and microservices.

service is implemented in a single binary, to large graphs of single-concerned and loosely-coupled *microservices*.[1,10] This shift is becoming increasingly pervasive, with large cloud providers, such as Amazon, Twitter, Netflix, Apple, and EBay having already adopted the microservices application model, and Netflix reporting more than 200 unique microservices in their ecosystem, as of the end of 2016.[1]

The increasing popularity of microservices is justified by several reasons. First, they promote composable software design, simplifying and accelerating development, with each microservice being responsible for a small subset of the application's functionality. The richer the functionality of cloud services becomes, the more the modular design of microservices helps manage system complexity. They similarly facilitate deploying, scaling, and updating individual microservices independently, avoiding long development cycles, and improving elasticity. For applications that are updated on a daily basis, modifying, recompiling, and testing a large monolith is both cumbersome and prone to bugs. Figure 1 shows the deployment differences between a traditional monolithic service, and an application built with microservices. While the entire monolith is scaled out on multiple servers, microservices allow individual components of the end-to-end application to be elastically scaled, with microservices of complementary resources bin-packed on the same physical server. Even though modularity in cloud services was already part of the service-oriented architecture (SOA) design approach, the fine granularity of microservices, and their independent deployment create hardware and software challenges different from those in traditional SOA workloads.

Second, microservices enable programming language and framework heterogeneity, with each tier developed in the most suitable language, only requiring a common API for microservices to communicate with each other; typically over remote procedure calls (RPC) or a RESTful API. In contrast, *monoliths* limit the languages used for development, and make frequent updates cumbersome and error-prone.

Finally, microservices separate failure domains across application tiers, allowing cleaner error isolation, and simplifying correctness and performance debugging, unlike in monoliths, where resolving bugs often involves troubleshooting the entire service. This also makes them applicable to Internet-of-Things (IoT) applications that often host mission-critical computation.

Despite their advantages, microservices represent a significant departure from the way cloud services are traditionally designed, and have broad implications in both hardware and software, changing a lot of assumptions current warehouse-scale systems are designed with. For example, since dependent microservices are typically placed on different physical machines, they put a lot more pressure on high bandwidth and low latency networking than traditional applications. Furthermore, the dependencies between microservices introduce backpressure effects between dependent tiers, leading to cascading QoS violations that propagate and amplify through the system, making performance debugging expensive in both resources and time.[11]

Given the increasing prevalence of microservices in both cloud and IoT settings, it is imperative to study both their opportunities and challenges. Unfortunately most academic work on cloud systems is limited to the available open-source applications; monolithic designs in their majority. This not only prevents a wealth of interesting research questions from being explored, but can also lead to misdirected research efforts whose results do not translate to the way real cloud services are implemented.

DeathstarBench SUITE

Our article,[10] presented at ASPLOS'19, addresses the lack of representative and open-source benchmarks built with microservices, and

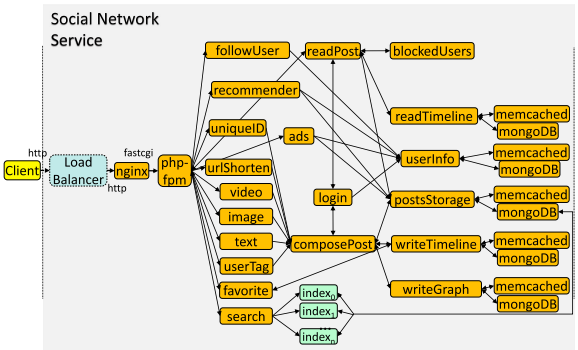

Figure 2. Graph of microservices in *Social Network*.

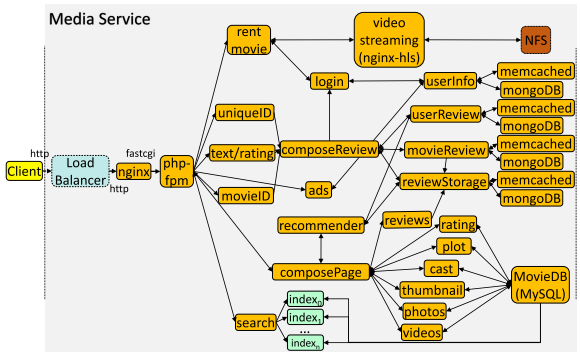

Figure 3. Graph of microservices in *Media Service*.

quantifies the opportunities and challenges of this new application model across the system stack.

Benchmark Suite Design: We have designed, implemented, and open-sourced a set of end-to-end applications built with interactive microservices, representative of popular production online services using this application model. Specifically, the benchmark suite includes a social network, a media service, an ecommerce shop, a hotel reservation site, a secure banking system, and a coordination control platform for UAV swarms. Across all applications, we adhere to the design principles of *representativeness*, *modularity*, *extensibility*, *software heterogeneity*, and *end-to-end operation*.

Each service includes tens of microservices in different languages and programming models, including node.js, Python, C/C++, Java, Javascript, Scala, and Go, and leverages open-source applications, such as NGINX, memcached, MongoDB, Cylon, and Xapian. To create the end-to-end services, we built custom RPC and RESTful APIs using popular open-source frameworks like Apache Thrift, and gRPC. Finally, to track how user requests progress through microservices, we have developed a lightweight and transparent to the user distributed tracing system, similar to Dapper and Zipkin that tracks requests at RPC granularity, associates RPCs belonging to the same end-to-end request, and records traces in a centralized database. We study both traffic generated by real users of the services, and synthetic loads generated by open-loop workload generators.

Applications in DeathStarBench

Social Network: The end-to-end service implements a broadcast-style social network with unidirectional follow relationships. Figure 2 shows the architecture of the end-to-end service. Users (client) send requests over http, which first reach a load balancer, implemented with nginx. Once a specific webserver is selected, also in nginx, the latter uses a php-fpm module to talk to the microservices responsible for composing and displaying posts, as well as microservices for advertisements and search engines. All messages downstream of php-fpm are Apache Thrift RPCs. Users can create posts embedded with text, media, links, and tags to other users. Their posts are then broadcasted to all their followers. Users can also read, favorite, and repost posts, as well as reply publicly, or send a direct message to another user. The application also includes machine learning plugins, such as user recommender engines, a search service using Xapian, and microservices to record and display user statistics, e.g., number of followers, and to allow users to follow, unfollow, or block other accounts. The service's backend uses memcached for caching, and MongoDB for persistent storage for posts, profiles, media, and recommendations. The service is broadly deployed at our institution, currently servicing several hundred users. We also use this deployment to quantify the tail at scale effects of microservices.

Media Service: The application implements an end-to-end service for browsing movie information, as well as reviewing, rating, renting, and streaming movies. Figure 3 shows the architecture of the end-to-end service. As with the social network, a client request hits the load balancer, which distributes requests among multiple nginx webservers. Users can search and browse information about movies, including their plot,

photos, videos, cast, and review information, as well as insert new reviews in the system for a specific movie by logging into their account. Users can also select to rent a movie, which involves a payment authentication module to verify that the user has enough funds, and a video streaming module using nginx-hls, a production nginx module for HTTP live streaming. The actual movie files are stored in NFS, to avoid the latency and complexity of accessing chunked records from nonrelational databases, while movie reviews are kept in memcached and MongoDB instances. Movie information is maintained in a sharded and replicated MySQL database. The application also includes movie and advertisement recommenders, as well as a couple auxiliary services for maintenance and service discovery, which are not shown in the figure.

E-Commerce Site: The service implements an e-commerce site for clothing. The design draws inspiration, and uses several components of the open-source Sockshop application. The application front-end in this case is a node.js service. Clients can use the service to browse the inventory using catalogue, a Go microservice that mines the back-end memcached and MongoDB instances holding information about products. Users can also place orders (Go) by adding items to their cart (Java). After they log in (Go) to their account, they can select shipping options (Java), process their payment (Go), and obtain an invoice (Java) for their order. Finally, the service includes a recommender engine for suggested products, and microservices for creating an item wishlist (Java), and displaying current discounts.

Hotel Reservation: The service implements a hotel reservation site, where users can browse information about hotels and complete reservations. The service is primarily written in Go, with the backend tiers implemented using memcached and MongoDB. Users can filter hotels according to ratings, price, location, and availability. They also receive recommendations on hotels they may be interested in.

Banking System: The service implements a secure banking system that processes payments, loan requests, and credit card transactions. Users interface with a node.js front-end, similar to the one in *E-commerce* to login to their account, search information about the bank, or contact a representative. Once logged in, a user can process a payment, pay their credit card bill, browse information about loans or request one, and obtain information about wealth management options. Most microservices are written in Java and Javascript. The back-end databases use memcached and MongoDB instances.

IoT Swarm Coordination: Finally, we explore an environment where applications run both on the cloud and on edge devices. The service coordinates the routing of a swarm of programmable drones, which perform image recognition and obstacle avoidance. We have designed two version of this service. In the first, the majority of the computation happens on the drones, including the motion planning, image recognition, and obstacle avoidance, with the cloud only constructing the initial route per-drone, and holding persistent copies of sensor data. This architecture avoids the high network latency between cloud and edge, however, it is limited by the onboard resources. In the second version, the cloud is responsible for most of the computation. It performs motion control, image recognition, and obstacle avoidance for all drones, using the ardrone-autonomy, and Cylon libraries, in OpenCV and Javascript, respectively. The edge devices are only responsible for collecting sensor data and transmitting them to the cloud, as well as recording some diagnostics using a local node.js logging service. In this case, almost every action suffers the cloud-edge network latency, although services benefit from the additional cloud resources. We use 24 programmable Parrot AR2.0 drones, together with a backend cluster of 20 two-socket, 40-core servers.

Adoption

DeathStarBench is open-source software under a GPL license.* The project is currently in use by several tens of research groups both in academia and industry. In addition to the open-source project, we have also deployed the social network as an internal social network at Cornell University, currently used by over 500 students, and have used execution traces for several

*https://github.com/delimitrou/DeathStarBench.

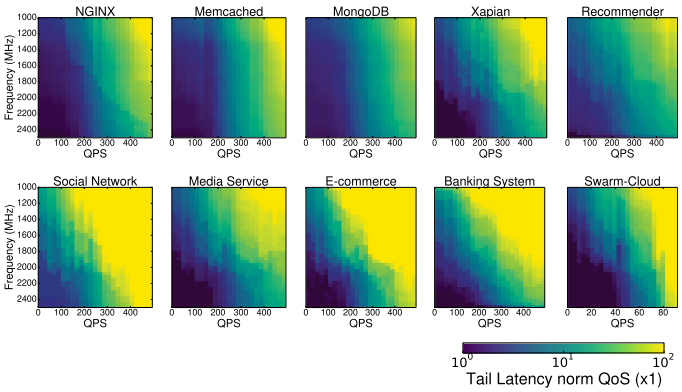

Figure 4. Tail latency with increasing load and decreasing frequency (RAPL) for traditional monolithic cloud applications, and the five end-to-end DeathStarBench services. Lighter colors (yellow) denote QoS violations.

research studies, including using ML in root cause analysis for interactive microservices.[11]

HARDWARE AND SOFTWARE IMPLICATIONS OF MICROSERVICES

We have used DeathStarBench to quantify the implications microservices have in cloud hardware and software, and what these implications mean for computer engineers. Below we summarize the main findings from our study.

Server Design

We first quantified how effective current *datacenter architectures* are at running microservices, as well as how datacenter hardware needs to change to better accommodate their performance and resource requirements. There has been a lot of work on whether small servers can replace high-end platforms in the cloud.[3] Despite the power benefits of simple cores, interactive services still achieve better latency in servers optimized for single-thread performance. Microservices offer an appealing target for simple cores, given the small amount of computation per microservice. Figure 4 (top row) shows the change in tail latency as load increases and frequency decreases using running average power limit (RAPL) for five popular, open-source single-tier interactive services: nginx, memcached, MongoDB, Xapian, and Recommender. We compare these against the five end-to-end services (bottom row).

As expected, most interactive services are sensitive to frequency scaling. Among the monolithic workloads, MongoDB is the only one that can tolerate almost minimum frequency at maximum load, due to it being I/O-bound. The other four single-tier services experience increased latency as frequency drops, with Xapian being the most sensitive, followed by nginx, and *memcached*. Looking at the same study for microservices reveals that, perhaps counterintuitively, they are much more sensitive to poor single-thread performance than traditional cloud applications, despite the small amount of per-microservice processing. The reasons behind this are the strict, microsecond-level tail latency requirements of individual microservices, which put more pressure on low and predictable latency, emphasizing the need for hardware and software techniques that eliminate latency jitter. Out of the five end-to-end services (we omit Swarm-Edge, since compute happens on the edge devices), the *Social Network* and *E-commerce* are most sensitive to low frequency, while the *Swarm* service is the least sensitive, primarily because it is bound by the cloud-edge communication latency, as opposed to compute speed.

Networking and OS Overheads

Microservices spend a large fraction processing network requests of RPCs or other RESTful APIs. While for traditional monolithic cloud services only a small amount of time goes toward network processing, with microservices this time increases to 36.3% on average, and on occasion over 50% of the end-to-end latency, causing the system's resource bottlenecks to change drastically. To this end, we also explored the potential hardware acceleration has to address the network requirements of microservices for low latency and high throughput network processing. Specifically, we use a bump-in-the-wire setup, seen in Figure 5(a), and similar to the one given by Firestone *et al.*[9] to offload the entire TCP stack on a Virtex 7 FPGA using Vivado HLS. The FPGA is placed between the NIC and the top of rack switch, and is connected to both with matching transceivers, acting as a filter on the network. We maintain the PCIe connection between the host and the FPGA for accelerating other services, such as the machine learning models in the recommender engines, during periods of low network load. Figure 5(b) shows the speedup

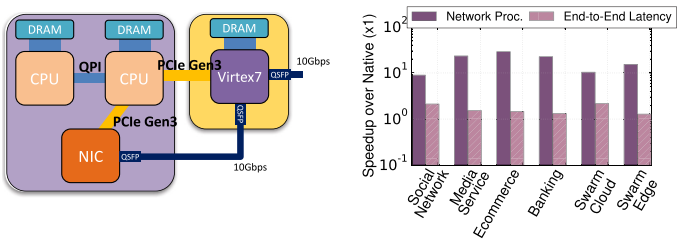

Figure 5. (a) Overview of the FPGA configuration for RPC acceleration, and (b) the performance benefits of acceleration in terms of network and end-to-end tail latency.

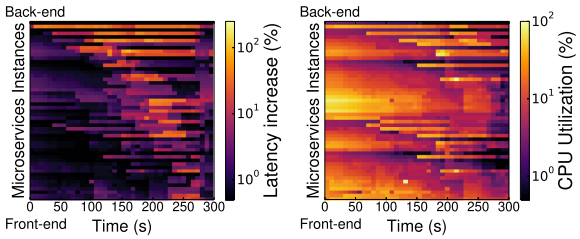

Figure 7. Cascading QoS violations in *Social Network* compared to per-microservice CPU utilization.

from acceleration on network processing latency alone, and on the end-to-end latency of each of the services. Network processing latency improves by $10-68x$ over native TCP, whereas end-to-end tail latency improves by 43% and up to $2.2x$. For interactive, latency-critical services, where even a small improvement in tail latency is significant, network acceleration provides a major boost in performance.

Cluster Management

A major challenge with microservices has to do with cluster management. Even though the cluster manager can elastically scale out individual microservices on-demand instead of the entire monolith, dependencies between microservices introduce backpressure effects and cascading QoS violations that propagate through the system, hurting quality of service (QoS). Backpressure can additionally trick the cluster manager into penalizing or upsizing a highly utilized microservice, even though its saturation is the result of backpressure from another, potentially not-saturated service. Not only does this not solve the performance issue, but can on occasion make it worse, by admitting more traffic into the system. The more complex the dependence graph between microservices, the more pronounced such issues become. Figure 6 shows the microservices dependence graphs for three major cloud service providers, and for one of our applications (*Social Network*). The perimeter of the circle (or sphere surface) shows the different microservices, and edges show dependencies between them. Such dependencies are difficult for developers or users to describe, and furthermore, they change frequently, as old microservices are swapped out and replaced by newer services.

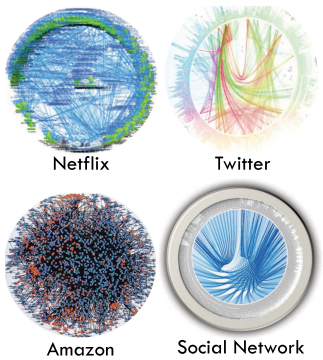

Figure 6. Microservices graphs for three production clouds, and our Social Network.

Figure 7 shows the impact of cascading QoS violations in the *Social Network* service. Darker colors show tail latency closer to nominal operation for a given microservice in Figure 7(a), and low utilization in Figure 7(b). Brighter colors signify high per-microservice tail latency and high CPU utilization. Microservices are ordered based on the service architecture, from the back-end services at the top, to the front-end at the bottom. Figure 7(a) shows that once the back-end service at the top experiences high tail latency, the hotspot propagates to its upstream services, and all the way to the front-end. Utilization in this case can be misleading. Even though the saturated back-end services have high utilization in Figure 7(b), microservices in the middle of the figure also have even higher utilization, without this translating to QoS violations.

Conversely, there are microservices with relatively low utilization and degraded performance, for example, due to waiting on a blocking/synchronous request from another, saturated tier. This highlights the need for cluster managers that account for the impact dependencies

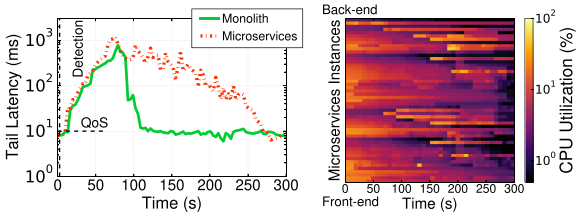

Figure 8. (a) Microservices taking longer than monoliths to recover from a QoS violation, even (b) in the presence of autoscaling mechanisms.

between microservices have on end-to-end performance when allocating resources.

Finally, the fact that hotspots propagate between tiers means that once microservices experience a QoS violation, they need longer to recover than traditional monolithic applications, even in the presence of autoscaling mechanisms, which most cloud providers employ. Figure 8 shows such a case for *Social Network* implemented with microservices, and as a monolith in Java. In both cases, the QoS violation is detected at the same time. However, while the cluster manager can simply instantiate new copies of the monolith and rebalance the load, autoscaling takes longer to improve performance. This is because, as shown in Figure 8(b), the autoscaler simply upsizes the resources of saturated services—seen by the progressively darker colors of highly utilized microservices. However, services with the highest utilization are not necessarily the culprits of a QoS violation, taking the system much longer to identify the correct source behind the degraded performance and upsizing it. As a result, by the time the culprit is identified, long queues have already built up, which take considerable time to drain.

Serverless Programming Frameworks

Microservices are often used interchangeably with serverless compute frameworks. Serverless enables fine-grained, short-lived cloud execution, and is well-suited for interactive applications with ample parallelism and intermittent activity. We evaluated our end-to-end applications on AWS's serverless platform, AWS Lambda and showed that despite avoiding the high costs of reserved idle resources, and enabling more elastic scaling in the presence of short load bursts, serverless also results in less predictable performance for interactive microservices for three reasons. First, on current serverless platforms communication between dependent *functions* (serverless tasks) happens via remote persistent storage, *S3*, in AWS's case. This not only introduces high, but also unpredictable latency, as S3 is subject to long queueing delays and rate limiting. Second, because VMs hosting serverless tasks are terminated after a given amount of idle time, when new tasks are spawned, they need to reload any external dependencies, introducing nonnegligible overheads. Finally, the placement of serverless tasks is up to the cloud operator's scheduler, and hence prone to long scheduling delays, and unpredictable performance due to contention from external jobs sharing system resources. Overall, we show that while similar, microservices and serverless compute each present different system challenges, and are well suited for different application classes.

Tail at Scale Effects

Finally, we explore the system implications of microservices at large scale. Tail at scale effects are well-documented in warehouse-scale computers,[4] and refer to system (performance, availability, efficiency) issues that specifically arise due to a system's scale. In this article, we use the large-scale deployment of the social network application with hundreds of real users to the impact of cascading performance hotspots, request skews, and slow servers on end-to-end performance.

Figure 9(a) shows the performance impact of dependencies between microservices on 100 EC2 instances. Microservices on the *y*-axis are again ordered from the back-end in the top to the front-end in the bottom. While initially all microservices are behaving nominally, at $t = 260$ s the middle tiers, and specifically composePost, and readPost become saturated due to a switch routing misconfiguration that overloaded one instance of each microservice, instead of load balancing requests across different instances. This in turn causes their downstream services to saturate, causing a similar waterfall pattern in per-tier latency to the one in Figure 7. Toward the end of the sampled time ($t > 500$ s) the back-end services also become saturated for a similar reason, causing microservices earlier in the critical path to saturate. This is especially evident for microservices

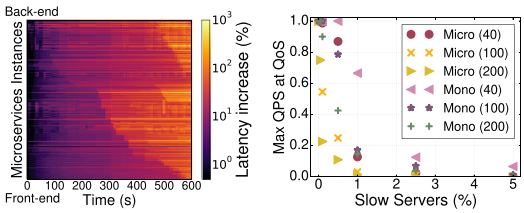

Figure 9. (a) Cascading hotspots in the large-scale *Social Network* deployment, and tail at scale effects from slow servers.

in the middle of the *y*-axis (bright yellow), whose performance was already degraded from the previous QoS violation. To allow the system to recover, we employed rate limiting, which constrains the admitted traffic, until hotspots dissipate. Even though rate limiting is effective, it affects user experience by dropping a fraction of requests.

Finally, Figure 9(b) shows the impact of a small number of slow servers on overall QoS as cluster size increases. We purposely slow down a small fraction of servers by enabling aggressive power management, which we already saw is detrimental to performance. For large clusters (>100 instances), when 1% or more of servers behave poorly, QPS under QoS is almost zero, as these servers host at least one microservice on the critical path, degrading QoS. Even for small clusters (40 instances), a single slow server is the most the service can sustain and still achieve some QPS under QoS. Finally, we compare the impact of slow servers in clusters of equal size for the monolithic design of *Social Network*. In this case, QPS is higher even as cluster sizes grow, since a single slow server only affects the instance of the monolith hosted on it, while the other instances operate independently. The only exception are back-end databases, which even for the monolith are shared across application instances, and sharded across machines. If one of the slow servers is hosting a database shard, all requests directed to that instance are degraded. The more complex an application's microservices graph, the more impactful slow servers are, as the probability that a service on the critical path will be slowed-down increases.

DeathStarBench draws attention to the need for research in the emerging application model of microservices and highlights the research areas with the highest potential for impact, while also providing a widely adopted open-source infrastructure to make that research possible and reproducible.

As with the hardware and cluster management implications above, these results again emphasize the need for hardware and software techniques that improve performance predictability at scale without hurting latency and resource efficiency.

LESSONS FROM DEATHSTARBENCH

DeathStarBench draws attention to the need for research in the emerging application model of microservices and highlights the research areas with the highest potential for impact, while also providing a widely adopted open-source infrastructure to make that research possible and reproducible.

Cloud systems have attracted an increasing amount of work over the past five to ten years. As cloud software evolves, the direction of such research efforts should also evolve with it. Given the increasing complexity of cloud services, the switch to a fine-grained, multitier application model, such as microservices, will continue to gain traction, and requires more attention from the research community in both academia and industry. DeathStarBench highlights the need for systems research in this emerging field, and quantifies the most promising research directions, and their potential impact.

DeathStarBench also highlights the need for hardware–software codesign in cloud systems, as applications increase in scale and complexity. Specifically, we showed that current platforms are not well-suited for the emerging cloud programming models, which require lower latency, more elastic scaling, and more predictable responsiveness. We demonstrated that microservices come with both opportunities and challenges across the system stack, and that for system designers to improve QoS without sacrificing resource efficiency, they need to rethink the current cloud stack in a vertical way, from hardware design and networking, to cluster management and programming framework design. Finally, we also quantified the potential hardware acceleration has toward addressing the performance requirements of interactive microservices, and

showed that programmable acceleration can greatly reduce one of the primary overheads of multitier services; network processing.

As microservices continue to evolve, it is essential for datacenter hardware, operating and networking systems, cluster managers, and programming frameworks to also evolve with them, to ensure that their prevalence does not come at a performance and/or efficiency loss. Both DeathStarBench and the resulting study of the system implications of microservices are a call to action for the research community to further explore the opportunities and challenges of this emerging application model.

ACKNOWLEDGMENTS

We sincerely thank C. Kozyrakis, D. Sanchez, D. Lo, as well as the academic and industrial users of the benchmark suite, and the anonymous reviewers for their feedback on earlier versions of this article. This work was supported in part by an NSF CAREER award, in part by NSF grant CNS-1422088, in part by a Google Faculty Research Award, in part by a Alfred P. Sloan Foundation Fellowship, in part by a Facebook Faculty Research Award, in part by a John and Norma Balen Sesquicentennial Faculty Fellowship, and in part by generous donations from Google Compute Engine, Windows Azure, and Amazon EC2.

REFERENCES

1. "The evolution of microservices," 2016. [Online]. Available: https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference
2. L. Barroso, U. Hoelzle, and P. Ranganathan, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool: San Rafael, CA, USA, 2018.
3. S. Chen, S. Galon, C. Delimitrou, S. Manne, and J. F. Martinez, "Workload characterization of interactive cloud services on big and small server platforms," in *Proc. Int. Symp. Workload Characterization*, Oct. 2017, pp. 125–134.
4. J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56 no. 2, pp. 74–80, 2013.
5. C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Houston, TX, USA, 2013, pp. 77–88.
6. C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoSAware cluster management," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Salt Lake City, UT, USA, 2014, pp. 127–144.
7. C. Delimitrou and C. Kozyrakis, "HCloud: Resource-efficient provisioning in shared cloud systems," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2016, pp. 473–488.
8. C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer... In the cloud," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2017, pp. 599–613.
9. D. Firestone *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implementation*, 2018, pp. 51–66.
10. Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2019, pp. 3–18.
11. Y. Gan *et al.*, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Apr. 2019, pp. 19–33.
12. D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 450–462.

Yu Gan is currently working toward the Ph.D. degree with the School of Electrical and Computer Engineering, Cornell University, where he works on cloud computing and root cause analysis for interactive microservices. He is a student member of IEEE and ACM. Contact him at yg397@cornell.edu.

Yanqi Zhang is currently working toward the Ph.D. degree with the School of Electrical and Computer Engineering, Cornell University, where he works on cloud systems and resource management for interactive microservices. He is a student member of IEEE and ACM. Contact him at yz2297@cornell.edu.

Dailun Cheng is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at dc924@cornell.edu.

Ankitha Shetty is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at aas394@cornell.edu.

Priyal Rathi is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at pr348@cornell.edu.

Nayan Katarki is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at nk646@cornell.edu.

Ariana Bruno is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at amb633@cornell.edu.

Justin Hu is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at jh2625@cornell.edu.

Brian Ritchken is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at bjr96@cornell.edu.

Brendon Jackson is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at btj28@cornell.edu.

Kelvin Hu is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at sh2442@cornell.edu.

Meghna Pancholi is currently working toward the B.S. degree with the School of Computer Science, Cornell University. Contact him at mp832@cornell.edu.

Yuan He is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at yh772@cornell.edu.

Brett Clancy is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at bjc265@cornell.edu.

Chris Colen is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at cdc99@cornell.edu.

Fukang Wen is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at fw224@cornell.edu.

Catherine Leung is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at chl66@cornell.edu.

Siyuan Wang is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at sw884@cornell.edu.

Leon Zaruvinsky is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at laz37@cornell.edu.

Mateo Espinosa is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at me326@cornell.edu.

Rick Lin is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at cl2545@cornell.edu.

Zhongling Liu is currently working toward the M.Eng. degree with the School of Electrical and Computer Engineering, Cornell University. Contact him at zl682@cornell.edu.

Jake Padilla is currently working toward the M.Eng. degree with the School of Computer Science, Cornell University. Contact him at jsp264@cornell.edu.

Christina Delimitrou is currently an Assistant Professor with the School of Electrical and Computer Engineering, Cornell University, where she works on computer architecture and distributed systems. Her research interests include resource-efficient datacenters, scheduling and resource management with quality-of-service guarantees, emerging cloud and IoT application models, and cloud security. Delimitrou received the Ph.D. degree in electrical engineering from Stanford University. She is a member of IEEE and ACM. Contact her at delimitrou@cornell.edu.